Università Degli Studi Di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Tesi di Laurea

Valutazione dei Web Services come meccanismo di supporto per ambienti a skeleton

Candidata

Caterina Lascaro

Relatore

Prof. Marco Danelutto

Controrelatore

Prof.ssa Susanna Pelagatti

Anno Accademico 2006/2007

A mio padre e a mia madre

Desidero ringraziare sinceramente il prof. Marco Danelutto, per il suo fondamentale aiuto, la sua disponibilità e la *santa pazienza* che ha avuto verso di me in questi mesi.
Desidero ringraziare tutti gli amici conosciuti a Pisa, per i momenti lieti e spensierati passati insieme e per aver sopportato i miei numerosi momenti di stress.
Nel mio pensiero, gli amici lontani, sempre vicini.con il loro immenso affetto.
Un ringraziamento particolare va a Massimo, per il suo sostegno e per aver creduto sempre nelle mie capacità, al contrario di me che spesso sono scettica nei miei riguardi.
Ed infine, ma non per ultima, desidero ringraziare mia madre, per essere sempre presente nella mia vita, pronta a sostenermi e ad incoraggiarmi, in ogni momento della mia vita, sia in quelli difficili sia *purtroppo* in quelli facili.

Sommario

Elenco delle Illustrazioni

Capitolo 1

Introduzione

Il Web Service, secondo la definizione data dal W3C [1], è un sistema software progettato per l'interoperabilità fra diversi elaboratori collegati in una medesima rete.

La sua principale connotazione è quella di offrire un'interfaccia software WSD (Web Services Description), con cui altri sistemi interagiscono per mezzo di messaggi formattati in XML, racchiusi in un busta SOAP [2], e trasportati mediante gli usuali protocolli di Internet.

In altre parole, il sistema dei Web Services, utilizzando un set base di protocolli disponibili ovunque, ha standardizzato i meccanismi con cui macchine diverse possono interagire fra loro, fornendo dei servizi. Inoltre il sistema mantiene la possibilità di utilizzare protocolli sempre più avanzati e specializzati per effettuare compiti specifici.

Questo è il motivo per cui i Web Services hanno assunto una grande importanza e stanno diventando uno standard di fatto; infatti sono utilizzati da aziende, da ricercatori, da servizi commerciali e così via .

Conoscendo la flessibilità e la duttilità dei Web Services, si è pensato di studiare in questa tesi la possibilità di utilizzarli nel contesto del calcolo parallelo "stile Muskel" e di valutarne le prestazioni.

Muskel [3] è una libreria per la programmazione parallela che fornisce agli utenti gli skeleton, cioè pattern che rappresentano un modo di parallelizzare una computazione sequenziale, e che possono essere usati per implementare applicazioni parallele efficienti. In questa tesi ci ciamo concentrati sullo skeleton di tipo Farm.

Gli skeleton, in Muskel, sono implementati sfruttando la tecnologia macro dataflow. Il macro dataflow è un'implementazione software su macchine convenzionali del modello dataflow, che trasforma una computazione a skeleton in un grafo, applicando un ordinamento parziale tra le sue operazioni per determinare le relazioni di precedenza fra esse.

Si è costruito, quindi, un prototipo che tramite un servizio web permetta di effettuare dei calcoli di qualsiasi natura distribuendo il lavoro su più macchine collegate in rete. Successivamente è stato verificato il suo funzionamento, la tempistica, la scalabilità e l'efficienza nel interpretare istruzioni macro dataflow. Nella verifica dei risultati, è emerso che per grane medio/alte la scalabilità del prototipo è ottima, mentre per grane fini è molto lontana dai valori ottimali.

La tesi è strutturata in 6 capitoli.

Il Capitolo 2 ha come argomento un ampio excursus sui Web Services, sulla loro architettura con la descrizione di componenti e protocolli. Successivamente è stata fatta una rassegna degli strumenti che permettono lo sviluppo e la pubblicazione dei servizi, dettagliando quelli attinenti alla tesi.

Nel Capitolo 3 si parla diffusamente del prototipo, della sua elaborazione e delle caratteristiche peculiari che deve presentare per svolgere il compito assegnatogli.

Il Capitolo 4 descrive ampiamente l'implementazione del prototipo, dettagliando i vari elementi che lo costituiscono e motivando tutte le scelte implementative operate.

Si parla del servizio elaborato, con le operazioni di load e compute e le scelte per la memorizzazione della classe del calcolo e per l'esecuzione del calcolo stesso. Segue la descrizione del thread, una disamina approfondita di tutti gli aspetti del suo lavoro e delle problematiche che comporta, come le librerie di supporto. Successivamente si passa a parlare della classe di calcolo, della scelta fatta per il passaggio dei parametri, separando l'impostazione dello stato della classe dal calcolo vero e proprio. Il modello del prototipo termina con al descrizione del Dispatcher, del Collector e del main. Infine vengono descritte le scelte dovute gli strumenti.

Nel Capitolo 5, si fa la valutazione del prototipo, attraverso determinati dei test ben precisi. Si danno le definizioni di tutti i parametri valutativi per l'efficienza e la performance ed

Il Capitolo 5 ha per argomento come si valuta il prototipo attraverso determinati test e in quale ambiente essi sono stati condotti. Emerge quindi l'analisi dei risultati, che sono parametri indicativi dell'efficienza e della performance del prototipo.

Il Capitolo 6 mostra le conclusioni al lavoro della tesi e i possibili sviluppi futuri.

Capitolo 2

Web Services

In questo capitolo viene introdotto il concetto di servizio web, descrivendone il modello in tutte le sue parti e sottolineando tutte quelle informazioni necessarie ad una facile comprensione dell'argomento.
Viene, inoltre, presentata una rassegna di tutti gli strumenti utili per la creazione di servizi web, con particolare attenzione a quelli utilizzati per il prototipo.

2.1 Overview

In letteratura un servizio web è definito come segue:

> *"Il Web Service è un'interfaccia che descrive un insieme di operazioni, accessibili attraverso una rete mediante messaggistica XML [04]. Questo sistema software è studiato appositamente per supportare interazioni macchina-macchina in rete [05]."*

L'interfaccia del WS è espressa tramite una descrizione del servizio (WSD), che contiene tutte le informazioni necessarie per interagire con il servizio stesso: i

formati dei messaggi, i tipi dei dati, i protocolli di trasporto e i formati per la serializzazione.

Il WSD è scritto in un formalismo (Web Services Description Language - WSDL [06]) processabile da un calcolatore e basato sullo standard XML.

La comunicazione tra il Web Service e gli altri sistemi avviene attraverso lo scambio di messaggi in XML che seguono lo standard SOAP[1] .

L'utilizzo di XML fa sì che le funzionalità del servizio siano indipendenti dalla loro implementazione, favorendo lo sviluppo di applicazioni distribuite basate sui Web Services su sistemi diversi, sia dal punto di vista architetturale sia da quello software (*loose coupling*).

Inoltre, una delle caratteristiche più interessanti dei web services è la possibilità di utilizzarli per comunicazioni e scambi di informazioni automatici: non interazione tra persona e applicazione (come avviene nella navigazione sul web) bensì interazione tra applicazioni. La descrizione formale e standardizzata dei servizi web, infatti, consente la ricerca e l'utilizzo dei web services senza richiedere necessariamente l'intervento umano.

Dunque, i web service sono pensati per i programmi, non per i programmatori. Si deve precisare che voler utilizzare a mano i meccanismi e i protocolli per la costruzione dei servizi e dei clients comporta complicazioni e lungaggini, in quanto i frameworks mettono a disposizione del programmatore librerie o totalmente automatiche o eccessivamente dettagliate, non pratiche, prolisse e non intuitive, per cui è facile incorrere nell'errore.

[1] Inizialmente acronimo di **S**imple **O**bject **A**ccess **P**rotocol, più tardi **S**ervice **O**riented **A**rchitecture **P**rotocol, ora invece solo SOAP

2.1.1 Modello SOA

La tecnologia dei WS si basa sul modello SOA[2] [07], che è rappresentata dall'interazione fra tre ruoli: service provider, service broker e service requester. Le interazioni implicano le operazioni di pubblicazione, di ricerca e di bind. Il modello può essere rappresentato secondo lo schema in figura 2.1 .

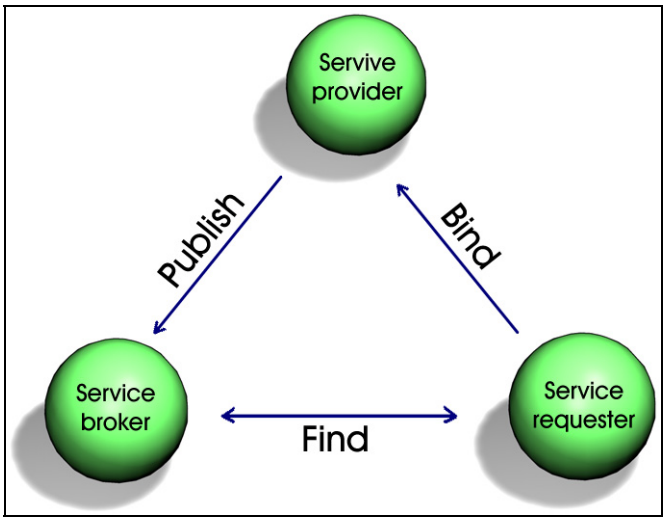

Figura 2.1 - Modello SOA

Le funzionalità dei tre ruoli a livello logico sono così indicate:

- **Serivice Provider** (sp): il proprietario del servizio o l'entità che si occupa di sviluppare ed offrire una serie di funzioni attraverso un'unica interfaccia omogenea.
- **Service Broker** (sb): un archivio in cui collezionare le descrizioni, ed eventualmente informazioni aggiuntive, su un servizio offerto da uno sp.
- **Service Requester** (sr): un'entità che chiede di utilizzare le funzioni offerte da un sp.

Per segnalare la disponibilità di un servizio, un sp invoca un'operazione di pubblicazione (publish) della descrizione verso un sb, quest'ultimo renderà

[2] **S**ervice **O**riented **A**rchitecture

disponibile tale informazione ai sr, che lo interrogheranno tramite un'operazione di ricerca (find). Una volta ottenuta la descrizione dell'interfaccia un generico sr potrà invocare, senza altri accordi, le funzioni offerte da un sp tramite un operazione di collegamento (bind).

Uno dei principali vantaggi introdotti da questo modello è il cosiddetto "**loose coupling**", cui abbiamo accennato sopra: per definire questo termine, ed i vantaggi ad esso correlati, [08] distingue due aspetti nelle relazioni che si instaurano tra sp e sr:

- **Dipendenza reale**: situazione in cui un sistema dipende dalle funzionalità fornite da un altro.
- **Dipendenza artificiale**: situazione in cui un sistema dipende dalle modalità e dagli strumenti tecnici per ottenere servizi forniti dalla controparte.

È chiaramente impossibile eliminare la dipendenza reale, mentre quella artificiale deriva dalle regole e dai relativi vincoli sull'uso di determinati strumenti o protocolli per accedere ad un servizio ed è quindi il fattore principale che incide sulla valutazione dell'efficienza ed efficacia di un modello d'integrazione. Quindi il loose coupling si ottiene quando la dipendenza artificiale è ridotta al minimo; pertanto il termine SOA può essere definito come uno stile di programmazione, che ha lo scopo di ottenere loose coupling tra software che interagiscono.

In generale, per raggiungere gli scopi prefissi dal modello SOA, vengono introdotti due vincoli nell' architettura:

- **Generalità delle interfacce**: le interfacce sono semplici perché contengono essenzialmente informazioni strutturate sulle operazioni offerte, senza descriverne la semantica

- **Messaggi descrittivi non prescrittivi**: i messaggi si limitano ad informare il destinatario circa le richieste effettuate, ma non indicano il comportamento, né il sottoprogramma da invocare per soddisfare una richiesta.

Un esempio pratico di applicazione di questo principio è dato dal web stesso: le risorse cambiano continuamente il proprio stato, i nodi cambiano le proprie interfacce, ma gli agenti continuano a scambiarsi rappresentazioni di risorse senza esserne influenzati.

A questo punto facciamo una puntualizzazione: non parleremo più di servizi ma di agenti.

La differenza è semplice: il servizio è la risorsa caratterizzata dall'insieme astratto di funzionalità, di cui è provvisto, implementata da un agente concreto, mentre l'agente è una parte concreta di software o hardware che invia e riceve messaggi.

Ci sono molti modi con cui un agente richiedente può *agganciare* e usare un WS, ma in generale sono necessari i seguenti passi, come illustrati nella figura 2.2

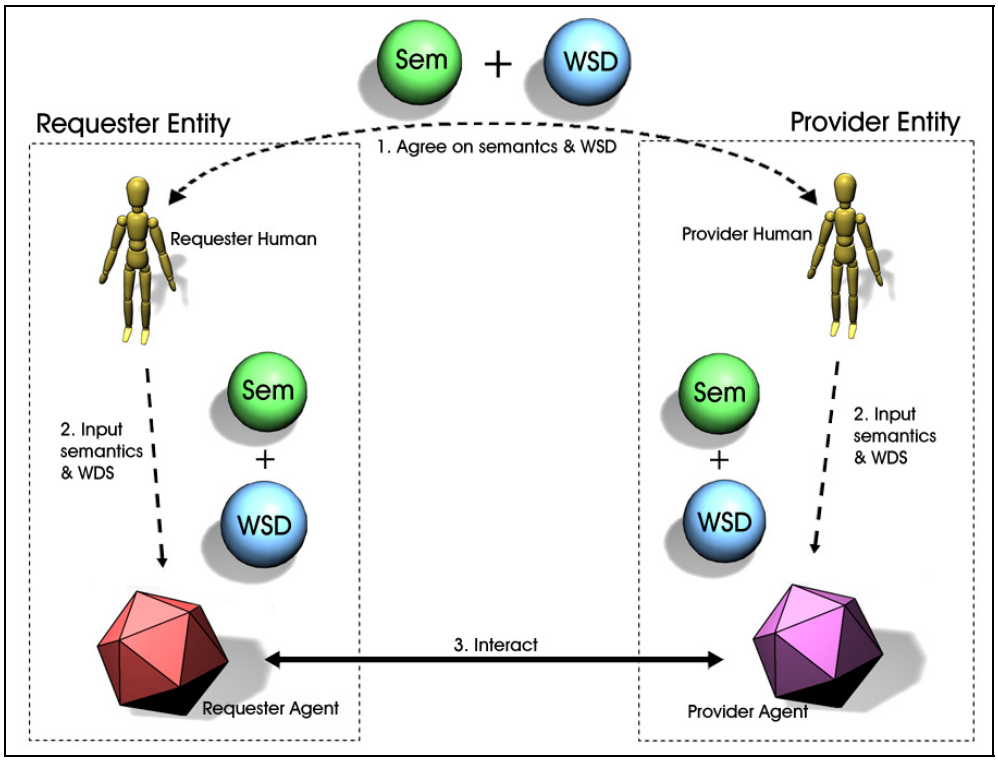

Figura 2.2 - Procedimento per "agganciare" un servizio web

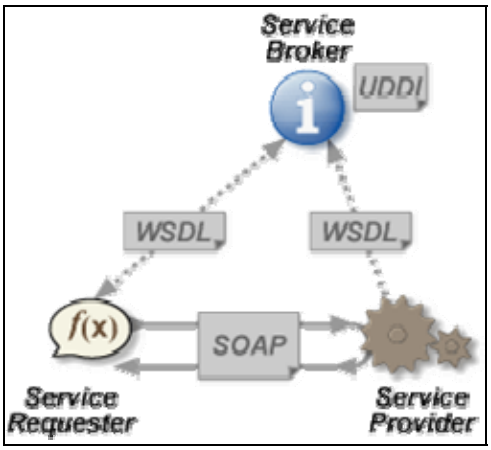

Figura 2.3 - Modello SOA e i protocolli usati nelle interazioni

1. Scambio dei messaggi: l'entità provider e l'entità requester concordano sulla semantica e sulla descrizione del servizio che regoleranno l'interazione;

2. Attuazione della semantica e descrizione del servizio da parte dei due agenti;
3. Interazione tra i due agenti requester e provider;

2.1.2 Architettura dei Web Services

La struttura dei Web Services è complessa e implica lo stratificarsi e l'interconnettersi di varie tecnologie. Esistono vari modi per visualizzare tali tecnologie, proprio come esistono varie maniere per costruire e usare i Web Services [27]. Una delle tante rappresentazioni è quella in figura 2.4 .

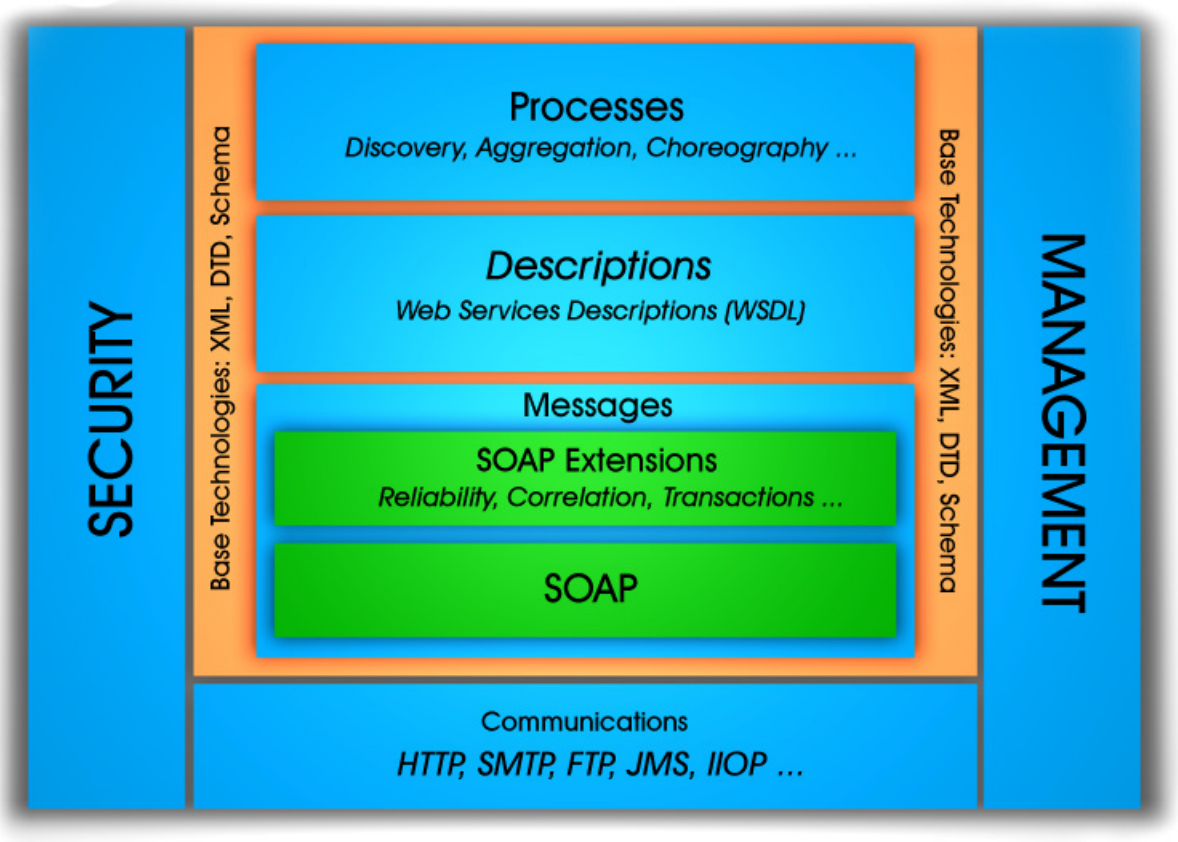

Figura 2.4 - Architettura dei Web Services

Nelle prossime sezioni, approfondiremo alcuni di questi aspetti.

XML

L' Extensible Markup Language (XML [09]) è un metalinguaggio universale, il cui scopo primario è quello di facilitare la condivisione di dati attraverso sistemi d'informazione differenti. Nel caso dei Web Services, l'XML soddisfa un requisito tecnologico chiave: grazie alla sua natura flessibile e intrinsecamente estensibile , XML riduce sensibilmente il carico della messa in opera di molte tecnologie, necessario per assicurare il successo dei Web Services.

Per la creazione di un documento XML valido, si devono seguire determinate regole, incluse in uno *schema XML*. L'unico schema XML che abbia per ora raggiunto la convalida ufficiale del W3C è l'**XML Schema**[12].

Come tutti i linguaggi di descrizione di contenuto XML, il suo scopo è delineare quali elementi sono permessi, quali tipi di dati sono ad essi associati e quale relazione gerarchica hanno fra loro gli elementi contenuti in un file XML. Un'altra cosa che XML Schema permette è l'estrazione, o meglio una visione, da un file XML di un insieme di oggetti con determinati attributi ed una struttura.

Le principali funzioni di XML Schema sono:

- Definire gli elementi (tag) che possono apparire in un documento
- Definire gli attributi che possono apparire in un elemento
- Definire quali elementi devono essere inseriti in altri elementi (child)
- Definire il numero degli elementi child
- Definire quando un elemento deve essere vuoto o può contenere testo, elementi, oppure entrambi
- Definire il tipo per ogni elemento e per gli attributi (intero, stringa, ecc, ma anche tipi personalizzati)
- Definire i valori di default o fissi per elementi ed attributi

Un piccolo esempio esplicativo di un documento XML è quello riportato di seguito.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<utenti>
    <utente>
        <nome>Luca</nome>
        <cognome>Ruggiero</cognome>
    </utente>
    <utente>
        <nome>Max</nome>
        <cognome>Rossi</cognome>
    </utente>
</utenti>
```

Dall'esempio è facile comprendere la struttura di un file XML:

- la prima riga indica la versione di XML in uso e specifica la codifica ISO per la corretta interpretazione dei dati;
- di seguito figura l'elemento radice che contiene tutti gli altri nodi del documento (il tag `<utenti>`)
- quindi seguono gli elementi annidati che racchiudono i dati da memorizzare

La struttura sopra riportata è un esempio di come sia possibile memorizzare i dati secondo uno schema ben preciso, dove il nome e il cognome di un utente sono rappresentati come due tag differenti e come figli di `<utente>`.

Un'alternativa a questo schema è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<utenti>
    <utente nome="Luca" cognome="Ruggiero" />
    <utente nome="Max" cognome="Rossi" />
</utenti>
```

La differenza risiede nel modo in cui viene rappresentato l'utente; infatti in quest'ultimo caso le informazioni non sono sono mostrate come tag separati, ma come proprietà del `<utente>` stesso.

SOAP

SOAP è un protocollo leggero per lo scambio di messaggi tra componenti software ed è alla base del protocollo di messaggistica dei Web Services. SOAP può muoversi sopra tutti i protocolli di Internet, ma soprattutto su HTTP che è il più comunemente utilizzato e l'unico ad essere stato standardizzato dal W3C.
Un vantaggio che si può ottenere usando SOAP sopra http, sia pure qualche volta discutibile, è il fatto che permette di effettuare comunicazioni oltrepassando proxies e firewall, più facilmente delle precedenti tecnologie.
Il procedimento per rappresentare i messaggi può essere paragonato a quello di una busta (**envelope**), codificata in XML, composta da un'intestazione (**header**) opzionale, e un contenitore (**body**), in cui inserire rispettivamente metadati e dati da comunicare; ogni busta è da considerarsi atomica, in quanto contiene al suo interno ogni informazione necessaria per interpretare i dati trasportati [13].
Le regole principali per realizzare un messaggio SOAP sono le seguenti:

- Deve essere codificato ovviamente con XML
- Deve utilizzare il SOAP Envelope namespace
- Deve utilizzare il SOAP Encoding namespace
- Non deve contenere il collegamento ad un DTD e non deve contenere istruzioni per processare XML

Lo stesso XML, però, comporta un appesantimento dei messaggi, in quanto sono presenti in essi molte informazioni aggiuntive e ripetitive e quindi SOAP risulta notevolmente più lento rispetto ad altre tecnologie, come CORBA.
Di seguito viene riportato un esempio di messagio SOAP, in cui un ipotetico cliente richiede informazioni su un prodotto da un immaginario warehouse web service:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productId>827635</productId>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

È facile comprendere anche in questo caso la struttura del messaggio, come per il documento XML. Considerato quanto detto finora, la sua interpretazione risulta banale.

WSDL

WSDL è una specifica per la creazione di documenti che descrivono i Web Services utilizzando la sintassi del XML Schema. WSDL esprime quattro tipi di dati:

- **Informazioni dell'interfaccia** che descrivono tutte le funzioni pubbliche disponibili
- **Informazioni dei tipi di dati** per tutte le richieste del messaggio e tutte le risposte del messaggio
- **Informazioni di binding** riguardanti il protocollo di trasporto da usare
- **Informazioni dell'indirizzo** per la locazione del servizio specificato

Tramite il WSDL, un client può localizzare un servizio web e invocare le funzioni disponibili. Con tools particolari è possibile inoltre rendere automatico questo processo, abilitando delle applicazioni per integrare nuovi servizi con poco o nessun codice scritto a mano.
Viene riportato di seguito un esempio di documento, estratto dalle specifiche WSDL 1.1.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           <element name="TradePrice">
              <complexType>
                  <all>
                      <element name="price" type="float"/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>
    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>
    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
   <binding name="StockQuoteSoapBinding"
            type="tns:StockQuotePortType">
      <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
      <soap:operation
            soapAction="http://example.com/GetLastTradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
   <service name="StockQuoteService">
      <documentation>My first service</documentation>
```

```
      <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
      </port>
   </service>
</definitions>
```

Nel documento si possono notare le seguenti sezioni:

- la sezione types definisce i tipi utilizzati all'interno dei messaggi del servizio;
- le sezioni message contengono le definizioni dei messaggi di scambio del servizio e fanno uso di parti definite come tipi nelle sezioni types;
- la sezione portType definisce le operazioni esposte dal Web Service. Per ogni metodo viene definito il messaggio di input ed il messaggio di output;
- la sezione binding contiene il collegamento tra il portType (cioè la definizione astratta del servizio) all'end-point fisico, indicando il protocollo da utilizzare e come ricondurre i messaggi di input ed output al protocollo utilizzato;
- la sezione service contiene la definizione del servizio in termini della sua descrizione e della posizione fisica del servizio (tipicamente il suo URL), definiti endpoint;

Discovery dei Web Services

Nel momento in cui un'entità requester decide di iniziare un'interazione con un'altra entità provider, ma non è a conoscenza di tutte le informazioni necessarie, il richiedente deve avviare una ricerca tramite il servizio di Discovery.

La Discovery è l'azione del localizzare una descrizione di un Web Service che può essere stata precedentemente non conosciuta e che incontra certi criteri funzionali. [10]

Un servizio di discovery è un ruolo logico, che può essere svolto sia dall'agente richiedente, sia dall'agente provider, sia da un altro agente.

Il protocollo maggiormente utilizzato è **UDDI** [14], che è un registry (ovvero una base dati ordinata ed indicizzata), basato su XML ed indipendente dalla piattaforma hardware, che permette alle aziende la pubblicazione dei propri dati e dei servizi offerti su internet.

Una "registrazione" UDDI consiste, infatti, di tre diverse componenti:

- **Pagine bianche** (White Pages): indirizzo, contatti (dell'azienda che offre uno o più servizi) e identificativi;
- **Pagine gialle** (Yellow Pages): categorizzazione dei servizi basata su tassonomie standardizzate;
- **Pagine verdi** (Green Pages): informazioni (tecniche) dei servizi fornite dall'azienda

Specifiche Aggiuntive

Esistono altre specifiche, alcune già funzionanti e altre ancora in via di sviluppo, per estendere e migliorare le capacità dei Web Services.

Le più importanti sono le seguenti:

- **il profilo della WS-I** (Web Service Interoperability Organization)[18], che è un insieme di specifiche per Web Service a dati livelli di revisione, unitamente ad un insieme di indicazioni d'implementazione e d'interoperabilità, segnalando come le specifiche possono essere usate per sviluppare i servizi interoperabili.

- **WS-Security** [16][17] definisce come usare l'XML Encryption e l'XML Signature in SOAP per far sì che lo scambio di messaggi sia sicuro, come alternativa o estensione ad HTTPS.
- **WS-Reliability** [19] è una specifica standard dell'OASIS che soddisfa condizioni per messaggi affidabili, in molte applicazioni di Web Services.
- **WS-ReliableMessaging** [20] è un protocollo per l'affidabilità dei messaggi tra due servizi web, messo a punto dalla Microsoft, dalla BEA e dalla IBM.
- **WS-Addressing** [21] è un modo per descrivere l' indirizzo del ricevente (e del mittente) di un messaggio, dentro il messaggio SOAP stesso.
- **WS-Coordination** [22] descrive un framework estendibile per fornire protocolli che coordinano le azioni di applicazioni distribuite. Questo protocolli di coordinazione sono usati per supportare un certo numero di applicazioni, inclusi quelli che necessitano di raggiungere un accordo consistente sull'esito delle transazioni distribuite.
- **WS-Transaction** [15] descrive i tipi di coordinazione che sono usati con il framework estendibile di coordinazione descritto nella specifica WS-Coordination. Definisce due tipi di coordinazione, **Atomic Transaction**, per le operazioni individuali e **Business Activity**, per le transazioni a lungo termine.
- **MTOM**[3] [23] è un metodo della W3C per inviare dati binari in maniera efficiente da e verso Web Service. Usa XOP[4] [24] per la trasmissione dei dati e sostituisce sia MIME che DIME.

[3] **M**essage **T**ransmission **O**ptimization **M**echanism

[4] **X**ML-binary **O**ptimized **P**ackaging è un metodo per includere efficientemente i dati binari in XML.

2.1.3 Aspetti critici dei Web Services

I Web Services presentano una larga diffusione per i numerosi aspetti positivi che li caratterizzano, ma come ogni tecnologia, pluristratificata e plurintegrata, non mancano di alcune negatività, per cui sono soggetti a cambiamenti e miglioramenti nel tempo per magnificarne i vantaggi e ridurne gli aspetti di minore agibilità.

I vantaggi di più immediato rilievo sono l'utilizzo di standard e protocolli "open" e di un formato di dati di tipo testuale: difatti, *per loro tramite* è possibile operare in ambienti multipiattaforma diversi tra di loro, è possibile far interoperare fra di loro software diversi, e accresce la portabilità e la scalabilità di componenti distribuiti.

Un vantaggio importante per creare e utilizzare Web Services è il "disaccoppiamento" fra sistema utente e Web Service, reso possibile dall'interfaccia standard pubblicata dal servizio. In questo modo è possibile rendere trasparenti modifiche all'una o all'altra delle applicazioni; questa flessibilità permette la creazione di sistemi software complessi, i cui componenti sono svincolati l'un dall'altro, e inoltre consente che codici e applicazioni già sviluppati possano essere riusabili.

Dato che è possibile usare HTTP su TCP sulla porta 80 come protocollo di trasporto, i Web Services hanno incontrato numerosi consensi; infatti tale porta è solitamente lasciata aperta dai sistemi firewall al traffico in entrata e in uscita dei sistemi aziendali, in quanto da tale porta passa il traffico HTTP dei browser web, e di conseguenza è possibile usare i Web Services senza apportare modifiche alle configurazione di sicurezza [25].

Considerando la cosa da un altro punto di vista, però, si nota che questa facilità nel oltrepassare il firewall è un baco di sicurezza, in quanto i firewall sono stati

ideati proprio per non permettere comunicazioni tra applicazioni esterne ed interne alle aziende, potenzialmente pericolose.

Una ragione che ha favorito l'adozione ed il proliferare dei Web Service è la mancanza, prima dello sviluppo di SOAP, di interfacce realmente funzionali per l'utilizzo di funzionalità distribuite in rete.

Le performance dei Web Services, invece, non sono concorrenziali rispetto ad altri metodi di comunicazione utilizzabili in rete. Questo svantaggio è legato alla natura stessa dei servizi web. Infatti essi sono basati su XML per cui ogni trasferimento di dati richiede l'inserimento di un notevole numero di informazioni supplementari (i tag XML) indispensabili per la descrizione dell'operazione. Inoltre c'è il problema della codifica e decodifica dei dati inviati ai capi della connessione. Queste due caratteristiche dei Web Service li rendono poco adatti a flussi di dati intensi o dove la velocità dell'applicazione rappresenti un fattore critico.

Inoltre ancora non esistono standard consolidati per applicazioni critiche quali, ad esempio, le transazioni distribuite.

2.2 Rassegna degli strumenti per lo sviluppo di Web Services

Il Web è assai ricco di framework e toolkit che assicurano al programmatore un valido supporto per la creazione e pubblicazione di servizi web, soprattutto facile da usare. Quelli più usati, comunque, sono i seguenti, tutti conformi al Basic Profile della WS-I [18]:

- Windows Communication Foundation della Microsoft
- Java Enterprise Edition

- Java Web Services Development Pack (WSDP) della SUN
- IBM WebSphere Application Server
- Apache Axis2

2.2.1 Windows Communication Foundation

Windows Communication Foundation (WCF) [26], precedentemente chiamato *Indigo*, è un sottosistema di comunicazione del framework .NET che consente di sviluppare ed eseguire sistemi distribuiti e interconnessi tra loro usando tutti i linguaggi supportati dal framework .NET (C#, C++, VB.NET, JScript.NET e ASP.NET). Riunisce, in una sola componente, tecnologie come Web Services, remoting[5], messaggistica e tutto quello che può rientrare nella categoria della comunicazione. WCF implementa molti WS* avanzati come WS-Addressing, WS-ReliableMessaging e WS-Security, ma [11] fa notare alcuni svantaggi dell'architettura .NET: .NET può solo girare su windows quindi ha una portabilità limitata.

2.2.2 Java Enterprise Edition

Questo è un ambiente molto più maturo per l'impresa [11]. J2EE ha un supporto maggiore dai più grandi commercianti di software come IBM, Oracle, HP e Sybase.

[5] *.NET Remoting* è una API Miscrosoft per le comunicazioni tra processi, presente già presente nella versione 1.0 di .NET Framework.

Java Enterprise Edition [28], in precedenza conosciuto con la sigla J2EE, consiste in un modello multilivello di programmazione per applicazioni, in API e definizioni di servizio, e in *reference implementations*[6].

È un piattaforma definita per specifiche, le implementazioni contengono estensioni particolari per i commercianti, la performance e le funzionalità disponibili variano molto tra i commercianti [29].

La specifica descrive i seguenti componenti [30]:

- Gli **Enterprise JavaBeans** definiscono un sistema a componenti distribuito e fornisce le tipiche caratteristiche richieste dalle applicazioni enterprise, come scalabilità, sicurezza, persistenza dei dati e altro.
- Il **JNDI**[7] definisce un sistema per identificare e elencare risorse generiche, come componenti software o sorgenti di dati.
- Il **JDBC**[8] è un'interfaccia per l'accesso a qualsiasi tipo di basi di dati.
- Il **JTA**[9] è un sistema per il supporto delle transazioni distribuite.
- Il **JAXP**[10] è un API per la gestione di file in formato XML.
- Il **Java Message Service** descrive un sistema per l'invio e la gestione di messaggi.

È divenuto uno standard, in quanto i provider devono attenersi a certi requisiti di conformità per poter affermare che i loro prodotti sono conformi a Java EE, anche senza la certificazione ISO[11] o ECMA[12]. Tra le applications server certificate vi sono *Sun Java System Application Server Platform Edition* 9.0,

[6] Una *reference implementation* è un'implementazione software di uno standard, che ha la funzione di aiutare nello sviluppo di altre implementazioni dello stesso standard.
[7] **J**ava **N**aming and **D**irectory **I**nterface
[8] **J**ava **D**ata**b**ase **C**onnectivity
[9] **J**ava **T**ransaction **A**PI
[10] **J**ava **A**PI for **X**ML **P**rocessing
[11] **I**nternational **O**rganization for **S**tandardization
[12] **E**uropean **C**omputer **M**anufacturers **A**ssociation

sviluppato come l'open-source server *GlassFish* e WebSphere Application Server (WAS) della IBM.

2.2.3 IBM WebSphere Application Server

IBM WebSphere Application Server (WAS) [31] è il prodotto principale all'interno del marchio WebSphere dell'IBM ed è costruito, appunto, usando standard open come J2EE, XML, e i Web Service.

Funziona con un gran numero di web server incluso Apache HTTP Server, Netscape Enterprise Server, Microsoft Internet Information Services (IIS), oltre a quelli della IBM stessa.

WAS implementa :

- WS-Security,
- WS-BusinessActivity,
- WS-ReliableMessaging,
- WS-Addressing,
- MTOM,
- il supporto per JAX-B[13] e StAX[14].

[13] **J**ava **A**rchitecture for **X**ML **B**inding

[14] **St**reaming **A**PI for **X**ML

2.2.4 Java Web Services Development Pack

Il Java Web Services Development Pack [32] è un SDK[15] gratuito per lo sviluppo di Web Service, web application e applicazioni java con le tecnologie più nuove per java

Allo stato attuale, il progetto GlassFish ha rimpiazzato le sue precedenti release per fornire nuovi tools per lo sviluppo di Web Service e XML tra le varie release di Java System Application Server, anche se per coloro che vogliono continuare ad usarlo nulla cambia.

JWSDP comprende le seguenti tecnologie [33] :

- JAX-WS (Java API for XML Web Services)
- XML Digital Signature
- XML and Web Services Security
- JAXB (Java Architecture for XML Binding)
- JAXP (Java API for XML Processing)
- JAXR (Java API for XML Registries)
- JAX-RPC (Java API for XML-based RPC)
- SAAJ (SOAP with Attachments API for Java)

2.2.5 Strumenti utilizzati

Gli strumenti scelti per questa tesi sono stati i seguenti:

- JDK 1.4.* / 1.5.*
- Apache - Tomcat v. 5.0.28
- Apache - Axis2/Java v. 1.1.1

[15] **S**oftware **D**evelopment **K**it – kit per lo sviluppo di software

JDK è l'acronimo di Java Development Kit [34] ed è un kit gratuito di sviluppo Java necessario alla programmazione di Applet ed altri programmi. Il software è fornito da Sun Microsystem. Sono state usate due versioni, in quanto il software è stato scritto originariamente con JDK 1.5.*, ma i test valutativi sono stati effettuati con lo stesso codice adattato al JDK 1.4.* a causa della versione installata nella macchina dove i test sono stati fatti; ai fini della tesi, questo era ininfluente.

Apache Tomcat [35] è un web container sviluppato presso la Apache Software Foundation (ASF). Tomcat implementa le specifiche Servlet e JavaServerPages (JSP) dalla Sun Microsystems, fornendo un ambiente per far girare codice java in collaborazione con un web server. Esso include anche le funzionalità di web server tradizionale, che corrispondono al prodotto Apache. Possiede tool per la configurazione e la gestione, ma è possibile fare ciò anche solo modificando alcuni file formattati in XML.
Rispetto alle versioni precedenti, Tomcat 5.x

- Implementa le specifiche Servlet 2.4 e JSP 2.0
- Riduce la garbage collection, migliora la performance e la scalabilità
- Possiede wrappers nativi per Windows e Unix per l'integrazione tra piattaforme
- Esegue il parsing JSP più veloce

Apache Axis2 [36] è il motore centrale per i Web Services. È stato completamente riprogettato e riscritto su Apache Axis [37] Soap Stack, che è assai diffuso.
Le implementazioni di axis2 sono disponibili in Java e in C.

Axis2 non fornisce solo la capacità di aggiungere interfacce di Web Service in applicazioni web, ma può anche funzionare come un applicazione server standalone.

È importante dare uno sguardo all'architettura di Axis2, poiché serve a comprendere meglio in quale contesto è stata sviluppata la tesi.

Le specifiche supportate da Axis2 nella versione 1.1.1 sono :

- SOAP 1.1 and 1.2
- MTOM , XOP and SwA[16]
- WSDL 1.1
- WS-Addressing
- WS-Policy
- SAAJ 1.1

È possibile usare come protocollo di trasporto HTTP, SMTP, JMS o TCP e supporta quattro data bindings[17]:

- ADB[18] [41]
- XMLBeans
- JibX [40]
- JaxMe[19] [39] and JAXBRI [42] (sperimentale)

Ed infine esistono due moduli aggiuntivi per ulteriori capacità quali, *Apache Rampart* [45] per WS-Security e *Apache Sandesha2* [43] per WS-Reliable Messaging.

[16] **S**OAP **w**ith **A**ttachments

[17] Il *Data Binding* è quel processo di rappresentazione delle informazioni in un documento XML come un oggetto nella memoria del computer. Un data binder realizza questo facendo un mapping tra gli elementi nello schema xml del documento e i membri della classe da rappresentare in memoria.

[18] **A**xis **D**ata **B**inding

[19] JaxME è un'implementazione open source di JAXB

Axis2 possiede un'architettura altamente modulare e flessibile, rendendo più facile il supporto delle specifiche dei Web Services come moduli. L'architettura di Axis2 ha adottato un metodo radicale da Axis per fornire maggiore scalabilità, una performance migliore e per meglio supportare i cambiamenti che avvengono nel panorama dei Web Services.
Essa presenta alcuni principi per preservare l'uniformità:

- *separa la logica dallo stato*: il codice che elabora non ha stato in Axis2. Ciò consente che il codice sia eseguito liberamente da thread paralleli.
- tutte le informazioni sono mantenute in un *modello d'informazione* permettendo di sospende e di ripristinare il sistema.

Axis2 Framework è costruito da moduli centrali, che insieme costituiscono il nucleo dell'architettura, e da altri moduli disposti intorno, come in figura 2.5.

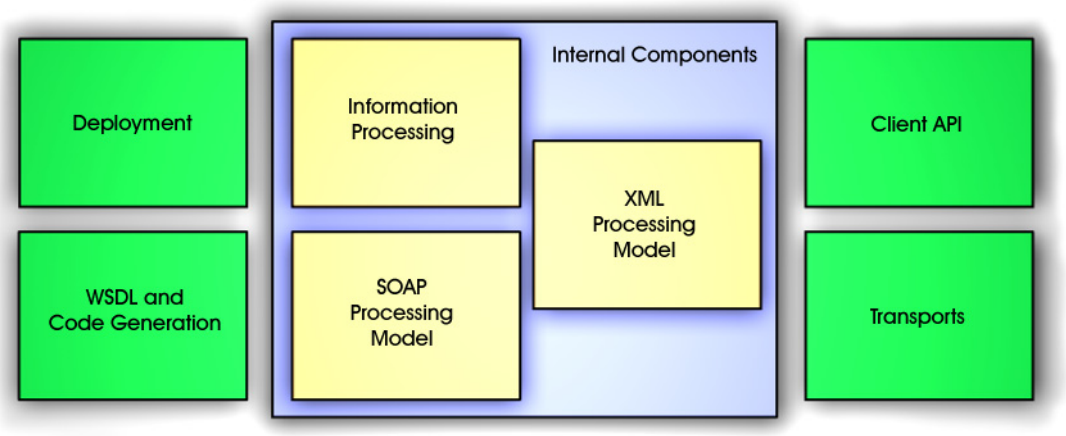

Figura 2.5 - Architettura modulare di Axis2

I moduli principali sono i seguenti:

- **Information Model** : Axis2 definisce un modello per maneggiare le informazioni e tutti gli stati sono contenuti in questo modello. Il modello

ha una gerarchia per le informazioni e il sistema gestisce il ciclo di vita degli oggetti in tale gerarchia.

- **XML processing Model** : maneggia messaggi SOAP che è il compito più importante e più complesso. Ha senso delegare questo lavoro ad un sottoprogetto separato, all'interno del progetto dei Web Services, permettendo che tale sottoprogetto (AXIOM o **AX**is **O**bject **M**odel)[38] fornisca delle API per l'insieme di informazioni di SOAP e XML e nasconderà le complessità del XML processing nell'implementazione.
- **SOAP Processing Model**: controlla l'esecuzione dell'elaborazione. Tale modello definisce le differenti fasi che l'esecuzione dovrà attraversare e l'utente può estendere questo modello in punti ben precisi.

Infine vi sono ulteriori moduli, non centrali:

- **Deployment Model:** permette all'utente di sviluppare servizi, configurare il trasporto, estendere il SOAP Processing model per il sistema, il servizio o la base di funzionamento.
- **Client API**: fornisce delle API per gli utenti, per far sì che possano comunicare con i Web Services tramite Axis2. C'è un insieme di classi per interagire con i MEP (Message Exchange Pattperns), sia stile IN-OUT sia IN-Only, che possono essere usate per costruire altri MEP.
- **Transports**: Axis2 definisce un framework di trasporto che permette all'utente di usare differenti protocolli di trasporto. I protocolli di trasporto si inseriscono in precisi punti del SOAP Processing Model. L'implementazione fornisce i più comuni protocolli di trasporto e l'utente può scriverne o aggiungerne di nuovi se e quando è necessario.

- **Code Generation**: Axis2 fornisce un tool per la generazione di codice che produrrà codice sia lato server sia lato client, con descrittori e test case[20].
- **Data Binding**: estende le Client API di base per renderle più convenienti per gli utenti.

Il Deployment Model fornisce meccanismi concreti per configurare Axis2 e possiede tre entità per fare ciò:

- il file **axis2.xml**: contiene la configurazione globale per il client e per il server e fornisce le seguenti informazioni:
  - i parametri globali
  - protocolli registrati di trasporto in entrata e in uscita
  - nomi delle fasi definite dagli utenti
  - moduli che sono globalmente "agganciati" (cioè a tutti i servizi)
  - i Message Receiver definiti globalmente
- l'archivio del servizio: deve contenere il file **META-INF/services.xml** e può contenere le classi dipendenti. Il file services.xml contiene le seguenti informazioni:
  - i parametri del servizio
  - i moduli "agganciati"
  - i Message Receiver specifici per il servizio
  - le operazioni del servizio
- l'archivio del modulo: deve contenere il file **META-INF/module.xml** e le classi dipendenti. Il file module.xml ha i parametri *Module* e le operazioni definite nel modulo stesso.

[20] Un *test case* è un insieme di condizioni o variabili per mezzo delle quali un test determinerà se i requisiti di un applicazione sono parzialmente o totalmente soddisfatti.

Nelle Client Api, vi sono tre parametri che decidono la natura dell'interazione del Web Service.

- Message Exchange Pattern (MEP)
- Il comportamento del trasporto, se è One-Way o Two-Way
- Il comportamento sincrono / asincrono delle client API

I MEP supportati sono due e sono In-Only e In-Out. L'implementazione è basata sulla classe chiamata ServiceClient e vi sono estensioni per ogni MEP che Axis2 supporta.

Il modulo per la generazione del codice ha adottato un metodo differente per svolgere il suo compito, rispetto ad Axis. Per prima cosa il cambiamento è nell'uso dei templates, detti XSL template che danno al generatore di codice la flessibilità per generare codice in più linguaggi. L'approccio base è di impostare il generatore per generare un XML (Emitter nella figura 2.6) tramite le informazioni estrapolate dal WSDL (CodeGeneration Engine nella figura 2.6) e analizzarlo(XSL T engine) con un template per generare il codice.

Per quanto riguarda il data binding, invece, esso è implementato in una maniera interessante. Non è incluso nella parte centrale dell'architettura deliberatamente e quindi il code generation consente di usare differenti data binding! Questo è ottenuto attraverso un meccanismo di estensione dove il CodeGeneration Engine chiama prima le estensioni e poi esegue l'emitter. Le estensioni popolano una mappa di QName contro i nomi delle classi che sono passate al CodeGeneration su cui l'emitter opera.

La figura 2.6 mostra tale struttura.

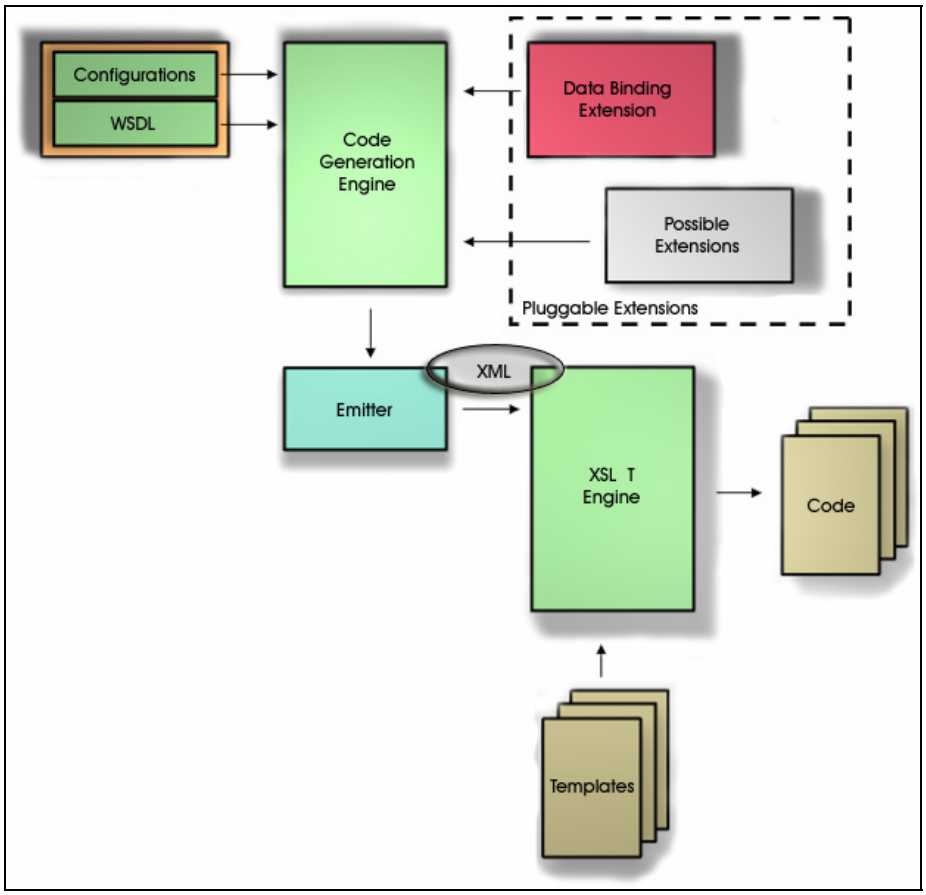

Figura 2.6 - Code Generation e Databinding

Sono disponibili i seguenti databinding

- ADB (Axis Data Binding): è un semplice framework che permette di compilare semplici schemi. È leggero e semplice, lavora su StAX ed è abbastanza performante. Tuttavia non supporta l'insieme completo di costrutti dello schema.
- XMLBeans: sostiene di supportare la specifica completa dello schema.
- JAX-Me e JAXB: sono stati aggiunti similarmente ad XMLBeans e servono come alternativa per l'utente, ma sono ancora in fase sperimentale.
- JibX: è la più recente aggiunta nella famiglia delle estensioni databinding, ma risulta un po' complessa nella configurazione.

Capitolo 3

Progetto Logico

3.1 Impostazione del problema

Come già accennato sopra, i vari livelli dell'architettura dei Web Services presentano come base tecnologica comune lo standard XML. Da ciò deriva che le funzioni del servizio possono essere implementate indipendentemente dal linguaggio di programmazione e dalle loro piattaforme hardware e software. Questo favorisce il loose coupling, argomentato nel Capitolo 2, l'interoperabilità del distributed computing e la scalabilità dei componenti distribuiti.

È a questo punto che scaturisce l'obiettivo della tesi, che è quello di analizzare la validità dei Web Services, quando siano applicati al calcolo parallelo nell'ambito del progetto Muskel e di valutarne l'efficienza.

Il lavoro viene diviso in due parti:

1. Elaborazione di un prototipo, che dimostri la fattibilità dell'utilizzo dei WS per la realizzazione del meccanismo "centrale" di Muskel, ovvero dell'interprete distribuito di istruzioni macro data flow
2. Valutazione dell'efficienza del prototipo

La tesi avrebbe potuto estendersi fino all'integrazione completa del prototipo con il progetto Muskel, ma per ragioni di tempo è stato necessario limitarsi agli esperimenti.
Sono stati scelti strumenti di facile e immediato utilizzo, di semplice installazione, evidenziando in questo modo la duttilità concettuale dei Web Services.

3.2 Modello del Prototipo

Per la realizzazione del prototipo, bisogna partire dall'analisi della sua funzione e delle caratteristiche che deve avere per realizzarla.
L'obiettivo della tesi viene raggiunto senza re-implementare Muskel nella sua interezza, pensando all'elaborazione di un prototipo che ricalchi la parte essenziale di Muskel stesso, e di utilizzare quindi una tecnologia differente.
Muskel è una libreria di programmazione parallela, basata su skeleton ed implementata usando il meccanismo dei grafi dataflow, preferendolo alla più tradizionale tecnologia a template. Muskel consente di programmare una computazione parallela mediante skeleton, in maniera semplice ed "user-friendly" . Gli skeleton messi a disposizione dalla libreria (pipeline, farm e loro composizioni arbitrarie), possono essere specializzati mediante parametri opportuni, come d'uso nell'ambito della programmazione a skeleton.
Una volta scritto il programma parallelo a skeleton ed indicati quali sono i task da eseguire, il meccanismo usato da Muskel per effettuare il calcolo risulta trasparente all'utente e viene riportato qui di seguito:

- Il programma a skeleton viene scomposto in "macro operazioni", riportate come nodi in un grafo dataflow, rispettandone le relazioni di

precedenza. Si deriva una macro operazione per ognuna delle porzioni di codice sequenziale utilizzate come parametri degli skeleton messi a disposizione da Muskel

- L'esecuzione del calcolo avviene attraverso l'elaborazione dei singoli nodi del grafo, da parte di entità preposte a questa funzione.
- Ciascuna delle macro operazioni derivate dal programma a skeleton viene inserita nella TaskPool che ha lo scopo di raggrupparli prima che venga eseguito il loro calcolo.
- Il calcolo dei singoli task (macro operazione con tutti i dati necessari alla loro elaborazione, ovvero istruzione "fireable" in terminologia data flow) avviene mediato da thread paralleli, i quali li raccolgono singolarmente i task dalla TaskPool e ne richiedono l'esecuzione ai nodi esecutori (locali o remoti) ad essi associati, depositandone successivamente la risposta ottenuta nella ResultPool.

La parte che ci siamo prefissi di re-implementare come prototipo in questa tesi è precisamente quest'ultima, dove i nodi esecutori sono implementati con Web Service.

Quindi, la funzione principale del nostro prototipo è quella di interpretare in remoto istruzioni macro dataflow di qualsiasi tipo di calcolo, sfruttando la flessibilità dei Web Services e mettendole in parallelo.

Il mandato della tesi è di utilizzare come base del prototipo il paradigma Farm, al posto di altri. Infatti, tale paradigma risponde a tutti i requisiti necessari alla realizzazione del prototipo stesso; in particolare, modella perfettamente la computazione master/slave che implementa il meccanismo di esecuzione remota di Muskel ed inoltre, i worker del Farm possono essere considerati concettualmente simili ai nodi esecutori.

Il *Farm* consente di parallelizzare una computazione, senza l'obbligo di conoscerne la struttura, tramite un insieme di worker tutti equivalenti, worker che devono sottoporre alla stessa computazione un insieme di task. Il modulo preposto allo scheduling dei task è l'Emitter, che li invia ai worker che si sono resi liberi. L'Emitter riconosce i worker liberi attraverso una coda asincrona di input con la quale riceve gli indici dei worker liberi. Dopo aver eseguito il calcolo, il worker invia il risultato al Collector, che riceve i dati e può eventualmente ordinarli.

Nel modello del prototipo, riportato nella figura 3.1, la funzione dell'Emitter è assunta dal Dispatcher.

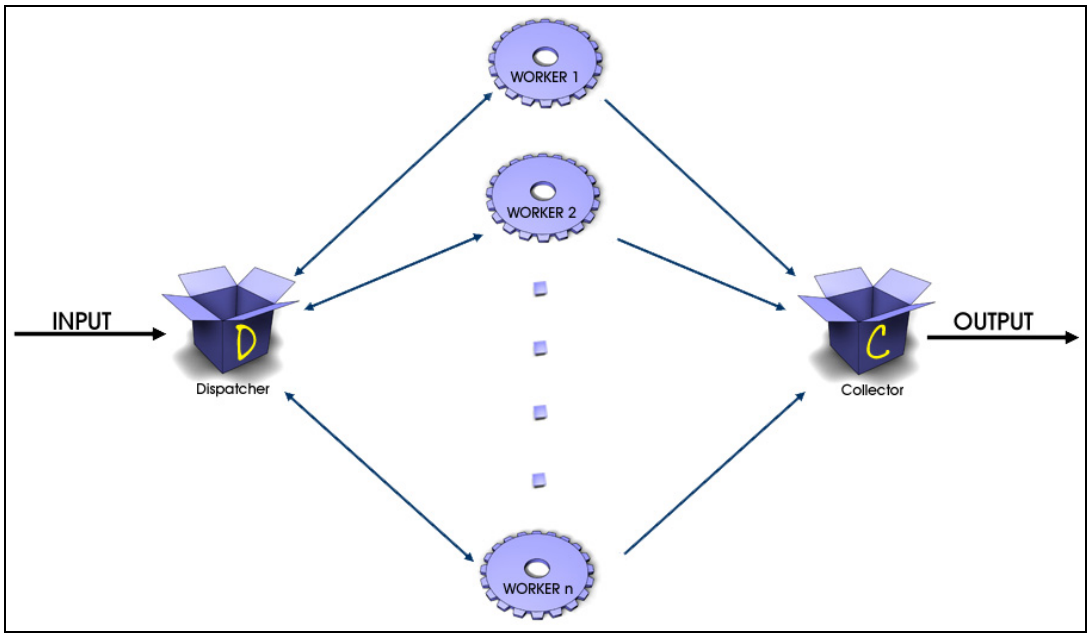

Figura 3.1 - Modello del Prototipo

I componenti del modello del prototipo sono:

- **Dispatcher** - unità la cui funzione è quella di prendere le procedure di calcolo e i dati di input dall'utente e di distribuirli ai Worker;

- **Collector** – unità che ha la funzione di conservare e raggruppare i dati di output ricevuti dai Worker e di restituirli all'utente;
- **Worker** - entità che richiedono le procedure di calcolo e i dati di input dal Dispatcher, effettuano il calcolo e inviano il risultato al collector.

La politica di distribuzione del lavoro ai Worker attribuita a questo prototipo è la politica FIFO (First In First Out): quando un Worker termina il proprio lavoro, si mette ordinatamente in fila pronto a ricevere il task successivo dal Dispatcher. La stessa politica, di converso, viene applicata nei riguardi del collector.

Dalla descrizione del nostro prototipo si evince che il nostro Dispatcher e il nostro Collector sono strettamente assimilabili alla TaskPool-Emitter e alla ResultPool-Collector rispettivamente di Muskel e Farm. Quanto al nostro tipo di Worker, esso si differenzia dal nodo esecutore perché utilizza la tecnologia del Web Service, mentre differisce dal Worker/Farm perché può eseguire qualsiasi tipo di calcolo.

Capitolo 4

Implementazione del prototipo

4.1 Schema d'implementazione

Una volta stabilita la struttura logica del prototipo, il passo successivo è l'elaborazione dello schema d'implementazione, come rappresentato nella figura 4.1 .

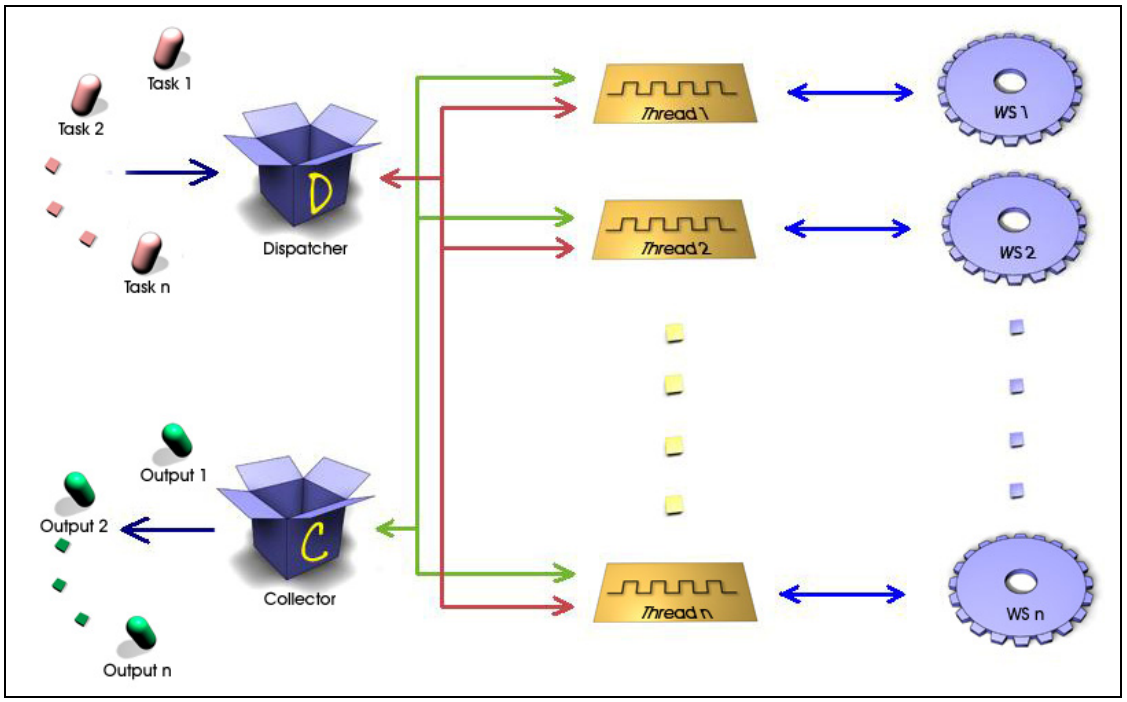

Figura 4.1 - Schema d'implementazione del prototipo

Il linguaggio di programmazione scelto è Java, in quanto il progetto Muskel è scritto in questo linguaggio. Gli strumenti scelti sono:

- Apache - Axis2/Java v. 1.1.1
- Apache - Tomcat v. 5.0.28
- JDK 1.5

Lo schema è composto da varie unità, che corrispondono a determinate classi java:

- Il Web Service (LoadAndCompute)
- Thread
- Dispatcher
- Collector
- La classe di calcolo

L'interazione tra le classi non si allontana molto da quella descritta nel modello del prototipo.
Il dispatcher riceve i dati di input dall'utente tramite il main e attende che i thread ne facciano richiesta.
I thread vengono avviati e informati dal main quale sia la classe di calcolo interessata, quindi inviano quest'ultima al servizio remoto ad essi associato.
Successivamente, essi fanno richiesta al dispatcher degli specifici parametri di calcolo da svolgere fino al loro esaurimento e rispettivamente li inviano al servizio, attendendone la risposta, per poi inviarla al collector.
Il collector attende i risultati dei calcoli dai thread, li conserva e li restituisce all'utente.
Infine il Web Service memorizza la classe di calcolo e quandone richiesto esegue il computo su uno specifico insieme di dati.

4.2 Il Web Service

Il Web Service LoadAndCompute è la componente principale di questo prototipo, in quanto la sua funzione è quella di eseguire i calcoli. Perché il prototipo sia facilmente gestibile, l'implementazione è stata ottimizzata inserendo un meccanismo per memorizzare i calcoli da eseguire.

Quindi il Web Service mette a disposizione dell'utente due operazioni: operazione di "*load*" e operazione di "*compute*".

La **load** permette di memorizzare la classe di calcolo in maniera permanente; infatti è stato scelto di memorizzare la classe di calcolo come file o archivio su disco, in modo da agevolarne il richiamo quando necessario.

I parametri dell'operazione sono:

- l'array di byte denominato `data`, che è la rappresentazione del file del calcolo nel tipo base
- la stringa `name`, per poter memorizzare il file con il giusto nome.

Il salvataggio avviene in una cartella, scelta durante la fase di sviluppo del codice e non modificabile in seguito, tramite la funzione `write` della classe `java.io.FileOutputStream`. Infine vengono effettuati dei test per verificare che la memorizzazione sia avvenuta correttamente, tramite due metodi della classe `java.io.File`, `exists` e `isFile`.

La **compute**, operazione che effettua il calcolo vero e proprio, deve per prima cosa caricare in memoria la classe del calcolo. Questo avviene utilizzando il metodo `loadClass` della classe `java.net.URLClassLoader`.

`URLClassLoader` pone subito la questione circa il valore del parametro richiesto dal costruttore, valore che dev'essere diverso, a seconda che la classe da caricare sia collocata in un *archivio* o semplicemente in un file "*.class*".

Infatti nel primo caso, `URLClassLoader` necessita dell' indirizzo assoluto del file, mentre nel secondo caso è sufficiente solo l'indirizzo della directory che lo contiene.

Questo problema è risolto grazie alla doppia interpretazione di uno dei parametri dell'operazione *compute*: la stringa `inJar`. Infatti, nel caso in cui la classe da caricare si trovi in un archivio, `inJar` deve contenere il nome completo dell'archivio stesso, mentre nel caso in cui essa si trovi nel suo file .class, `inJar` dovrà contenere la parola-chiave "*no*".

Il nome della classe da caricare è contenuto nella stringa `classe`, passata come parametro alla compute. La `loadClass` restituisce l'oggetto `Class` relativo, denominato `cls` .

Per il passaggio degli input al calcolo per i test da effettuare, è stato scelto di usare un'array di oggetti Long, esclusivamente per semplicità.

Per il calcolo dalla classe già caricata, si utilizza la Reflection, che è la capacità di un programma di eseguire elaborazioni che hanno per oggetto il programma stesso, e in particolare la struttura del suo codice sorgente.

Con la reflection si è in grado di esaminare il byte code della classe e di estrapolare da esso tutte le parti che costituiscono la classe stessa, tra cui i metodi e le loro firme.

In questa maniera è possibile impostare i dati di input e invocare il metodo del calcolo. Si è proceduti come di seguito: per mezzo dell'oggetto della classe `cls`, si "estrae" il metodo in discussione con il `getMethod`, il quale ha bisogno, come parametri, del nome del metodo e di un array di oggetti `Class`, che rappresentano i tipi dei parametri del metodo stesso.

In questa maniera si è ottenuti un oggetto `Method` che rappresenta un preciso metodo della classe caricata. Con questo oggetto è possibile, a questo punto,

invocare il metodo richiesto tramite `invoke`, i cui parametri sono un'istanza della classe caricata ed un array di Object, contenente i parametri necessari.
Ad elaborazione terminata, il risultato del calcolo viene restituito in un array di oggetti Long.
Associato ad ogni servizio, deve esservi il corrispettivo file service.xml con tutte le informazioni necessarie al suo deployment. Il file services.xml di LoadAndCompute contiene il nome completo della classe del servizio, l'abilitazione di MTOM e il Message Receiver usato per le operazioni, che è `RPCMessageReceiver` ed ha MEP in stile IN-OUT.

4.3 Il Thread

Il thread è il componente architetturale, che è in stretto collegamento con il servizio.
Prima di parlarne per esteso, bisogna introdurre, in senso generale, il "lato client" del Web Service, che invia al servizio il lavoro da svolgere e ne riceve i risultati.
Per fare ciò, si possono seguire due vie: utilizzare a mano le librerie di Axis2 oppure avvalersi dell'aiuto dello strumento *wsdl2java*, messo a disposizione dallo stesso Axis2, strumento che genera lo *stub* relativo ad un servizio specifico. Quest'ultima è la scelta fatta per questa tesi.
Uno stub nello sviluppo del software è una porzione di codice usata per sostituire alcune funzionalità di programmazione, in questo caso molto utile, in quanto permette al programmatore di non addentrarsi nel labirinto della libreria di Axis2 e di non incorrere in errore.

Lo strumento *wsdl2java* necessita di due parametri fondamentali: l'indirizzo del WSDL del servizio e il data binding da utilizzare per la conversione Java↔XML.

La scelta del data binding è ricaduta su ADB (Axis2 DataBinding), poiché fra tutti quelli disponibili, è quello che consente in maniera meno complicata la suddetta conversione.

L'implementazione dei data binding non consente, però, flessibilità al programmatore, poiché attua un associazione stretta solamente tra alcune classi Java e i rispettivi elementi XML. Questo accade con alcuni tipi base e le loro classi corrispondenti, con le stringhe, il calendario, la data, *array*, *Map* e con alcune sottoclassi di *Collection*.

Inoltre, c'è la possibilità che sia l'utente a crearsi una classe Java e la utilizzi come parametro del servizio. Poi, sarà il data binding a convertirla nell'elemento XML chiamato *complexType*. La conversione viene fatta, andando a convertire tutte le variabili e le strutture dati, presenti nella classe, ricorsivamente.

Nei restanti casi, il data binding fa un'associazione fra classi Java e l'elemento XML *anyType*. Nella conversione opposta l' *anyType* viene associato alla classe base della libreria usata, che nel caso del ADB è la classe `OMElement` della libreria AXIOM (AXis Object Model) . L'elemento *anyType* può essere presente anche nei figli del *complexType*.

Il problema, quindi, risiede nel fatto che in un Web Service, per poter utilizzare un parametro che appartenga ad una classe diversa da quelle predefinite in XMLSchema, è necessario essere in grado di trasformare un oggetto di tale classe in un oggetto OMElement. Purtroppo non esiste un metodo garantito e facile da utilizzare per fare ciò, ma solo classi e metodi il cui risultato può essere aleatorio.

Per risolvere suddetta e altre questioni, la SUN contribuisce per il completo sviluppo di JAXBri project, che come già detto nel Capitolo 2, ha l'obiettivo di fornire una tecnologia per la conversione Java↔XML che sia persistente.
Il thread riceve dal main all'atto della sua creazione tutte le informazioni necessarie per svolgere i suoi compiti:

- L'indirizzo del servizio associato (`String targetEndpoint`)
- Il nome della classe di calcolo o il suo file .class (`String cl`)
- Il nome dell'archivio che contiene la classe di calcolo o la parola-chiave "*no*" (`String fjar`)
- Il Dispatcher
- Il Collector
- Le variabili per impostare la classe di calcolo

Per comunicare con il suo Web Service, il thread sfrutta la libreria fornita dallo stub:

1. Invia al servizio il file contenente la classe di calcolo
2. Invoca il metodo `synchronized getNext` del Dispatcher al quale accede in mutua esclusione per ottenere i dati di input su cui lavorerà il servizio
3. Richiede al servizio l'esecuzione del calcolo e relativo risultato
4. Invia quest'ultimo al Collector tramite il metodo `setReturn`
5. Ripete sempre dal punto 2, fino ad esaurimento degli input

Per garantire che i dati di input e di output siano strettamente correlati, è stato necessaro introdurre dei task numerati.

4.4 La classe di Calcolo

Per classe di calcolo si intende quella classe che ogni utente deve implementare per eseguire il calcolo voluto.

La classe deve possedere due metodi:

- `setParam`
- `compute`

Il metodo setParam ha la funzione di passare i dati di input per il calcolo e di impostare lo stato della classe stessa, ovvero di inizializzare tutte le variabili e le strutture dati della classe.

Il metodo compute, invece, contiene la sequenza di operazioni che dovranno essere eseguite dal servizio.

Ai fini della tesi, è stato deciso di parametrizzare la classe di calcolo usata per i test allo scopo di raccogliere più procedure di calcolo, caratterizzate dalla dimensione del carico di input e dal tempo di esecuzione, evitando di replicare inutilmente il codice.

I due parametri sono

- **N**, la dimensione del carico
- **S**, il tempo di esecuzione

Il calcolo si basa sull'incremento degli elementi di un array e di un tempo di attesa S espresso in millisecondi. Il parametro S viene passato all'interno dell'array stesso, nell'ultima posizione. Infine per poter verificare il corretto uso della Reflection, è stata ideata un'altra classe di calcolo, a partire da quella appena descritta, immessa in package contenente una struttura di classi ideata appositamente e raggruppato il tutto in un archivio.

4.5 Dispatcher, Collector e il main

Il *Dispatcher* è un elemento architetturale la cui funzione è quella di distribuire tutti i task designati dal main al thread che ne fa richiesta.

Per simulare il reperimento dei dati di input, è stato creato un metodo che genera un array di N+1 elementi, di cui i primi N sono i dati su cui effettuare il calcolo e l'ultimo è il numero del task corrispondente. Sarà compito del thread correlare gli input agli output, come detto sopra, e sostituire al numero del task il tempo S.

Il *Collector* effettua il lavoro inverso al Dispatcher, ovvero raccoglie i dati di output in una struttura dati e li rende disponibili all'utente.

La classe *main*, chiamata Client, ha la funzione di raccogliere le informazioni sul calcolo e sui dati di input dati dall'utente, di avviare tutto l'iter e di stampare a video i risultati.

4.6 Aspetti legati alla scelta degli strumenti

Durante lo sviluppo del prototipo, sono state valutate alcune scelte in vari livelli :

- Invio dei file dal thread al servizio corrispondente.

  Axis2 supporta due meccanismi alquanto efficienti per questa operazione e sono MTOM e SwA (vedi Capitolo 2).

  La dimensione del file è ridotta e quindi si poteva scegliere di non usare nessun meccanismo, ma per rendere più efficiente l'operazione si è optato su MTOM, scartando SwA.

Infatti quest'ultimo non è stato ancora debitamente implementato e quindi non offriva garanzie; inoltre bisognava scrivere il codice senza usufruire dello stub.

Al contrario, Axis2 permette l'uso di MTOM, molto facile, sia al client sia al server:

- ❖ nel lato server, MTOM viene abilitato aggiungendo un tag nel file services.xml, indipendentemente dai settaggi del web container
- ❖ nel lato client, è sufficiente settare una proprietà nelle opzioni del ServiceClient

- Garantire al servizio sicurezza e affidabilità.

  Axis2 assolve questo compito mediante due moduli che implementano due specifiche WS*:

  - ❖ Il modulo *Apache Sandesha2* per quanto riguarda WS-ReliableMessaging
  - ❖ Il modulo *Apache Rampart* per quanto riguarda WS-Security

  Nella stesura del prototipo, dopo questa valutazione si è optato di non usare i suddetti moduli sia perché si voleva verificare in primis la fattibilità dell'intero approccio (WS per implementazione di un interpreter macro data flow distribuito) che per ragioni di tempo (l'implementazione del prototipo ha richiesto più tempo del previsto, soprattutto perché l'utilizzo della tecnologia WS "a mano" è risultato piuttosto pesante).

Grazie alla modularità di Axis2, non sono state necessarie ulteriori impostazioni, nei suoi file di configurazione.

Quando si è pensato di usare il prototipo in ambito non valutativo, è insorta la necessità di diversificare i tipi dei dati di input rendendoli non predefiniti.

Nel linguaggio Java, il Web Service vede questi dati come generici Object e come tali li passa al metodo setParam che ha il compito di castarli nei rispettivi tipi.

Idealmente questo sarebbe stato possibile se non si fossero verificati problemi a causa del data binding.

Come primo tentativo, è stato scelto il data binding ADB. Nella generazione dello stub, l'array di Object richiesto dal servizio si è trasformato in un array di OMElement.

Per ottenere la trasformazione di un oggetto qualsiasi nel corrispettivo OMElement, come accennato sopra, non vi è una utility che lo faccia automaticamente: farlo a mano si è rivelato oltremodo difficile senza garanzie di risultato.

Il tentativo successivo è stato fatto con il data binding JAXB, che permette di passare i parametri al servizio esattamente come sono stati scritti nel codice Java. Questo data binding è ancora in fase sperimentale, per cui si è verificato il problema che gli oggetti pervenuti al servizio perdevano le informazioni che li caratterizzavano e risultavano inutilizzabili.

La mancanza di tempo sufficiente ha impedito l'ulteriore approfondimento della questione.

Questo è il motivo per cui il servizio web di questo prototipo ha il limite di poter fare calcoli con dati di input aventi tutti un unico tipo.

Capitolo 5

Valutazione Sperimentale del prototipo

Dopo l'implementazione del prototipo, per giudicarne la convenienza, abbiamo provveduto a testarne l'efficienza e la performance.

I passi presi in considerazione sono:

- funzionamento del prototipo
- calcolo dei tempi di setup
- controllo della scalabilità
- calcolo dell'efficienza

Il primo approccio alla valutazione sperimentale del prototipo è quello di verificare il suo corretto *funzionamento*, quindi controllare se i risultati dei calcoli sono esatti.

Si è fatto stampare a video gli array di input dal dispatcher e i corrispondenti output, presi dal collector.

Il *tempo di setup* è il tempo trascorso dall'avvio dei thread fino alla ricezione del primo array di input da parte loro e varia al variare della dimensione del file contenente la classe di calcolo.

Il *tempo ideale* invece è il tempo che un programma impiega per effettuare lo stesso calcolo, in maniera sequenziale, senza thread ne Web Services.

La *scalabilità* di un programma parallelo fornisce la misura di quanto si riducono i tempi nell'esecuzione di un programma parallelo rispetto allo stesso programma eseguito su un'unica macchina in maniera sequenziale.

Per esprimere la scalabilità si divide il tempo di servizio del programma eseguito su un'unica macchina (T_s^1) per il tempo di servizio del programma parallelo (T_s^n), con grado di parallelismo n. La formula è $s(n) = \frac{T_s^1}{T_s^n}$.

La valutazione grafica della scalabilità si fa studiando i grafici, in cui viene mostrato il tempo di completamento ottenuto rispetto al tempo ideale di calcolo: se il tempo di completamento parallelo si avvicina molto al tempo ideale, allora è un buon risultato in termini di scalabilità.

Un concetto legato alla scalabilità è *l'efficienza*. Essa indica il grado di utilizzazione dei moduli che compongono il sistema parallelo e si ottiene dividendo la scalabilità per il grado di parallelismo. La formula è $\varepsilon(n) = \frac{s(n)}{n} = \frac{T_s^1}{n * T_s^n}$. Il massimo valore dell'efficienza è 1 e in tal caso anche la scalabilità è ottima.

Sia l'efficienza sia la scalabilità possono essere calcolate in tempi di completamento, usando le stesse formule appena descritte [46]. In questa tesi, il tempo di completamento è il tempo che trascorre dall'invio del primo array di input da parte del dispatcher fino alla ricezione dell'ultimo array di output da parte del collector. La formula è $T_c = T_s * num\_task$.

5.1 L'ambiente d'esecuzione

I test sono stati effettuati utilizzando un cluster di workstation chiamato `Pianosa` [45], situato presso il Dipartimento di Informatica dell'Università degli Studi di Pisa.

Il cluster è composto da 32 nodi più un nodo master adibito alle sole funzioni di amministrazione.

I 32 nodi hanno la stessa configurazione:

- 1GB di Memoria RAM
- 2 Hard-Disk da 18GB
- 1 processore Intel® Pentium® III Mobile CPU da 800MHz
- 32KB di Cache L1
- 3 Ethernet Pro 1000

Tutti i nodi, quindi, sono interconnessi per mezzo di tre reti Ethernet con banda 100 Mbits/sec. Una di queste reti è utilizzata per il traffico di sistema, specialmente per PVFS e NFS[21], mentre le altre due sono dedicate al traffico utente. Gli indirizzi dei nodi sono relativi alle tre schede di rete e per l'i-esimo nodo essi risultano come ui, ti e vi.

Per i test sono stati utilizzati 17 nodi, da u3 a u14 e da u16 a u20: il primo per il main e le altre 16 per il calcolo parallelo.

5.2 Test effettuati

Per poter toccare tutti i punti caratterizzanti la performance del prototipo, sono stati ideati tre tipi di test:

[21] PVFS: **P**aralel **V**irtual **F**ile **S**istem NFS: **N**etwork **F**ile **S**ystem

- **Test A**, sul tempo di Setup
- **Test B**, sul tempo di Comunicazione
- **Test C**, sul tempo di Completamento

I *test A* servono a valutare il tempo impiegato ad eseguire le dovute impostazioni del prototipo per poter effettuare il calcolo vero e proprio. Vengono condotti su 9 tipi di archivi java, differenti per dimensioni, costruiti a partire dall'archivio spiegato nel paragrafo 4.4 :

- 1,28 KB
- 7,7 KB
- 10 KB
- 50 KB
- 100KB
- 300 KB
- 600 KB
- 800 KB
- 1 MB

I *test B* servono a valutare i tempi di comunicazione (tempo che intercorre tra l'invio di un array da parte di un thread al servizio e l'inizio della *compute* da parte del servizio stesso) per le tre dimensioni di carico: 8, 16 e 1024.

I *test C*, infine, servono a valutare il tempo impiegato dal prototipo ad eseguire tutti i task designati e sono stati effettuati

- su tre dimensioni di carico (parametro N) corrispondenti alla dimensione degli array di input: 8, 16 e 1024
- su diversi tempi di calcolo (parametro S):
    - 10 msec,
    - 100 msec,
    - 500 msec,

  - 1000 msec,
  - 5.000 msec,
  - 10.000 msec
- con diversi gradi di parallelismo: 1, 2, 4, 8 e 16

Tutti i test sono stati condotti senta tenere conto del carico della CPU da parte di altri utenti e di conseguenza i test A e B sono stati ripetuti più volte e fra tutti i valori rilevati sono stati considerati solo i più piccoli.

5.3 Risultati ottenuti

5.3.1 Test A

I tempi di setup risultati possono essere valutati con lo studio dei tre grafici nelle figure 5.1, 5.2, 5.3.

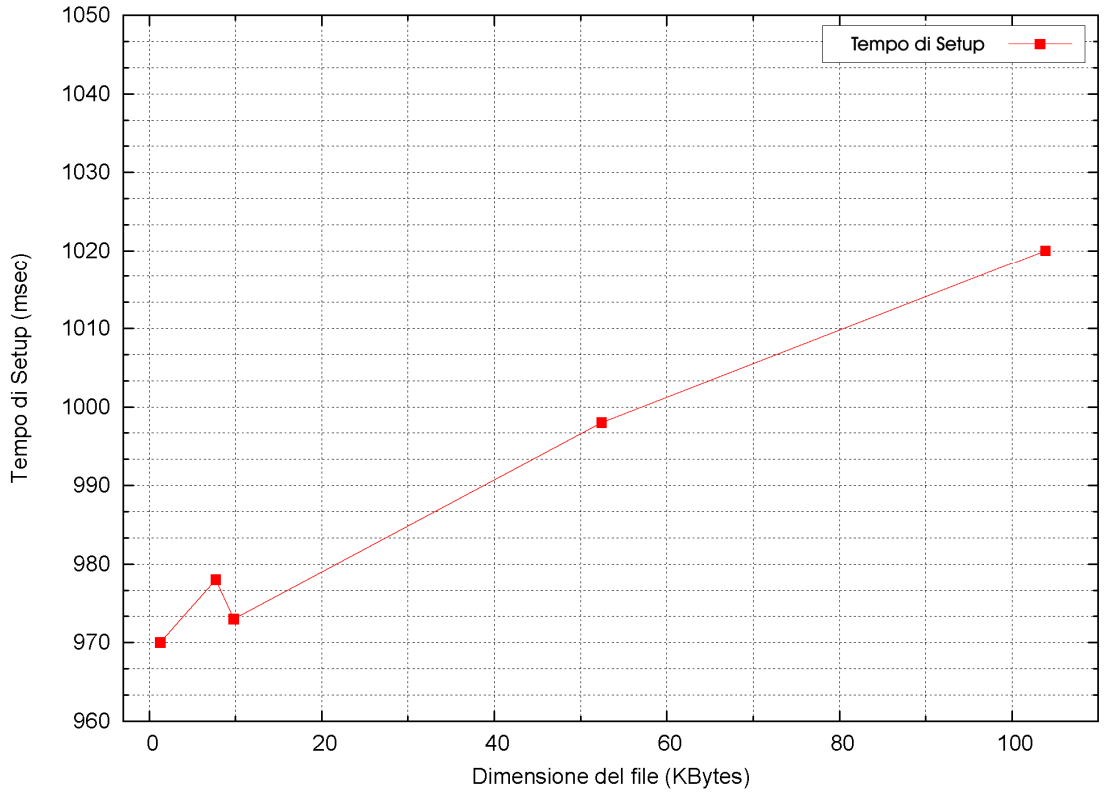

Figura 5.1 - Tempi di setup per file di dimensione inferiore ai 100KB

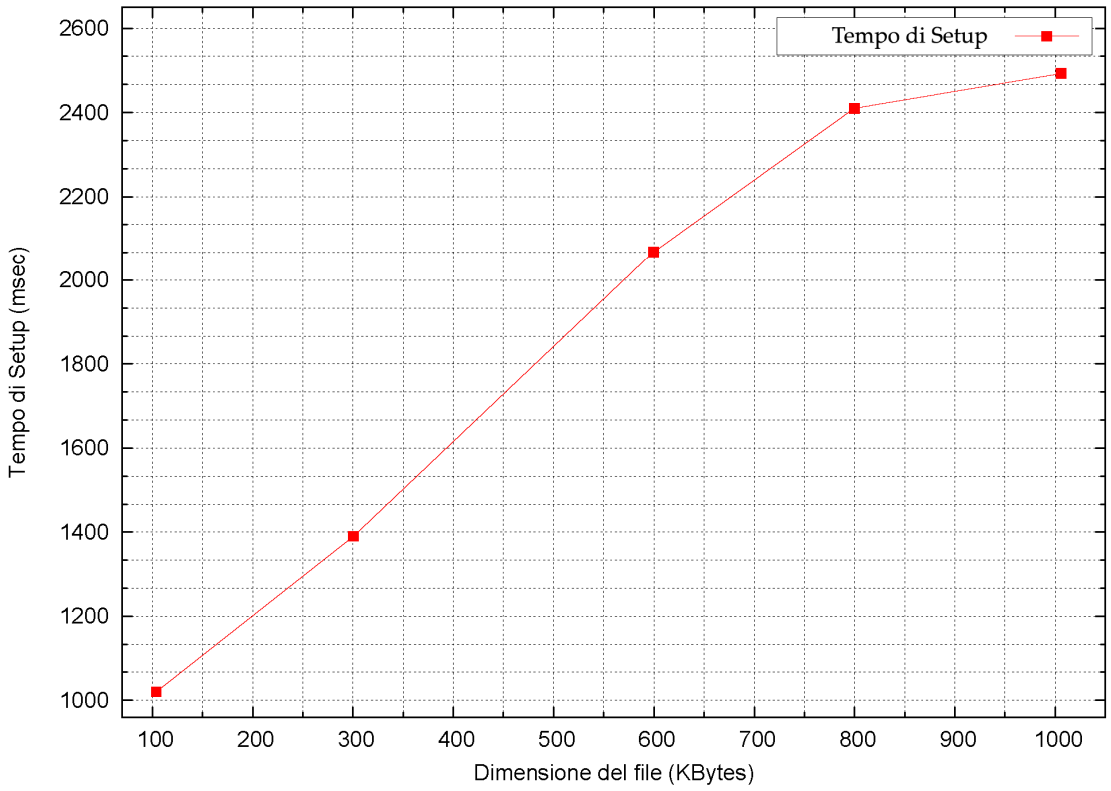

Figura 5.2 - Tempi di setup per file di dimensione superiore ai 100KB

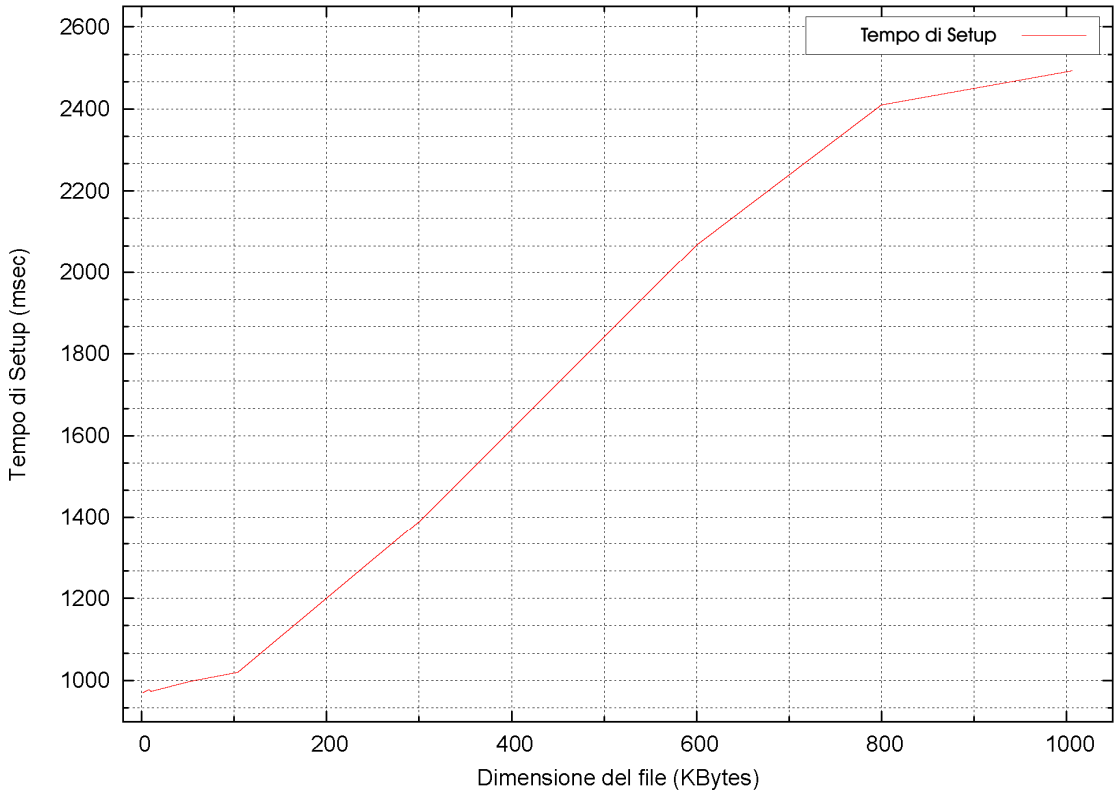

Figura 5.3 - Tempi di setup per file di dimensione a partire da 1KB fino ad 1MB

Era atteso che il tempo di setup fosse lineare rispetto alla dimensione del file, ovvero che il rapporto esistente fra la dimensione del file e il corrispondente tempo di setup rimanesse pressoché invariato. Al contrario, si è constatato che nei casi presi in esame la proporzione non è sempre costante (figura 5.2) e che per certi valori si sono avuti tempi di setup maggiori per file di più piccole dimensioni, come esemplificato in figura 5.1 per il file 7,7KB e per il file di 10KB. Questo, comunque, non va considerato in maniera negativa, in quanto l'incostanza di questi tempi non incide molto sulla valutazione del prototipo.

5.3.2 Test B

In questi test, sono stati calcolati i tempi di comunicazione fra i thread e i Web Service, per ogni dimensione di carico.

Essi sono risultati :

- ✓ *25,5 msec,* per un carico di 8 elementi
- ✓ *27,5 msec,* per un carico di 16 elementi
- ✓ *185,5 msec,* per un carico di 1024 elementi

Graficamente questo si mostra come nella figura 5.4 :

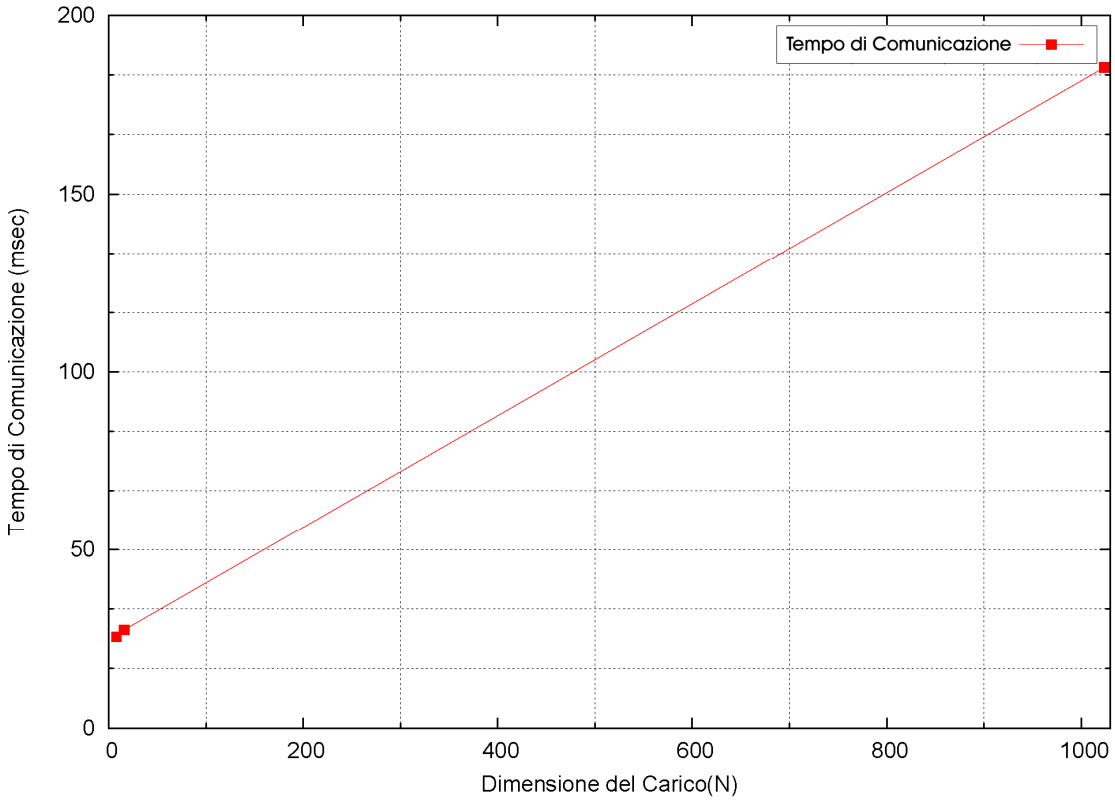

Figura 5.4 - Tempi di Comunicazione

5.3.3 Test C

Per una valutazione ottimale del tempi di completamento e di tutto ciò che ne consegue, i risultati sono stati suddivisi secondo la grana del calcolo.

Nel calcolo parallelo, la *grana* (g) indica la quantità di calcolo effettuato in relazione alla comunicazione ed è data proprio dal rapporto tra il tempo di calcolo e il tempo di comunicazione.

Quando la grana è *fine*, il tempo impiegato nelle comunicazioni diventa importante rispetto al tempo di completamento; il parallelismo a grana fine significa che i singoli task sono relativamente piccoli in termini di dimensione del codice e tempo di esecuzione. I dati sono trasferiti fra processori frequentemente in quantità di uno o più blocchi di memoria.

Quando la grana è *grossa*, avviene il contrario: i dati sono comunicati in maniera infrequente dopo più ampie quantità di calcolo.

Di conseguenza, più piccola è la granularità, più grande è il potenziale per il parallelismo e quindi lo speed-up[22], ma più grande è l'overhead della sincronizzazione e della comunicazione [47].

Grana Fine

Tra i vari scenari di questo tipo di test, hanno grana fine i seguenti:

- S = 10ms; N = 8
- S = 10ms; N = 16
- S = 10ms; N = 1024
- S = 100ms; N = 1024
- S = 500ms; N = 1024
- S = 1000ms; N = 1024

Considerando i tempi di comunicazione visti nel precedente paragrafo, si può notare che nei primi 4 scenari quest'ultimi sono di molto superiori ai tempi di calcolo, mentre nei restanti due, la differenza non è sensibile.

[22] Nel calcolo parallelo lo *speedup* indica quanto più veloce è un algoritmo parallelo rispetto al corrispondente algoritmo sequenziale.

È possibile valutare il tempo di completamento rispetto al tempo ideale con l'aiuto dei grafici nelle figure 5.5 – 5.10 . In tali figure, i tempi di completamento del prototipo sono segnati con dei quadrati di colore rosso.

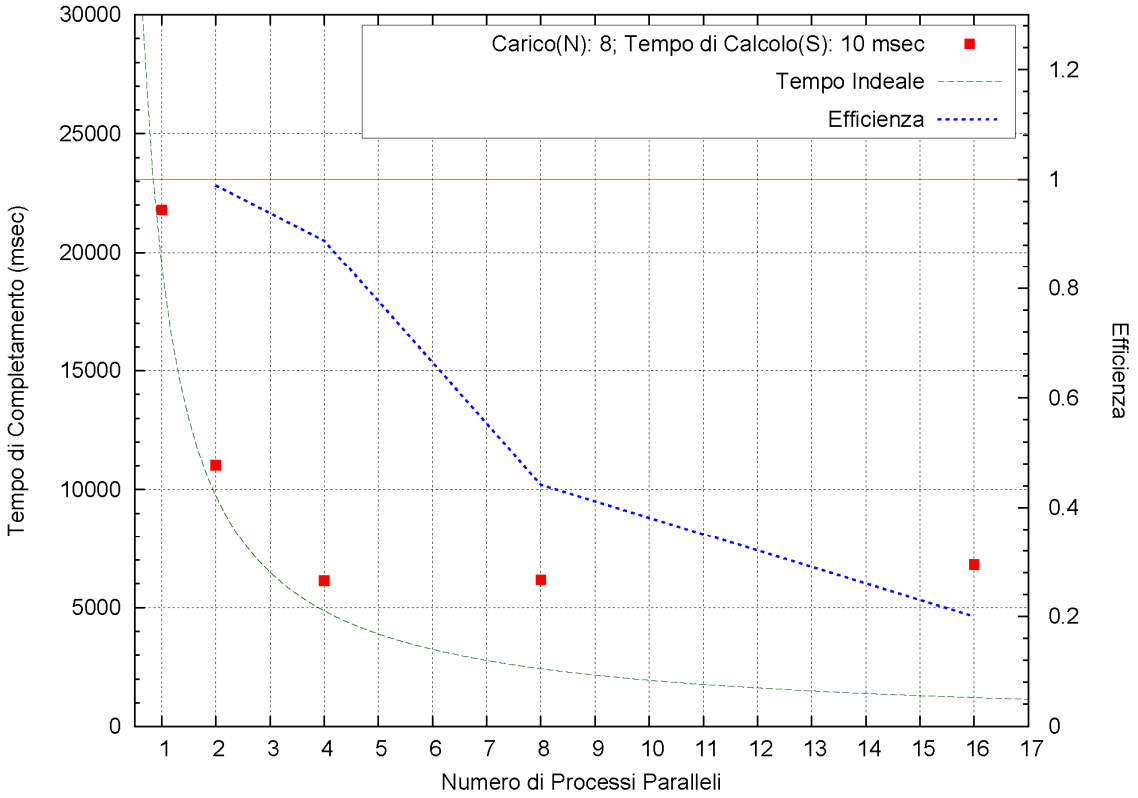

Figura 5.5 - Grana Fine (g= 0,3921); N = 8; S = 10msec

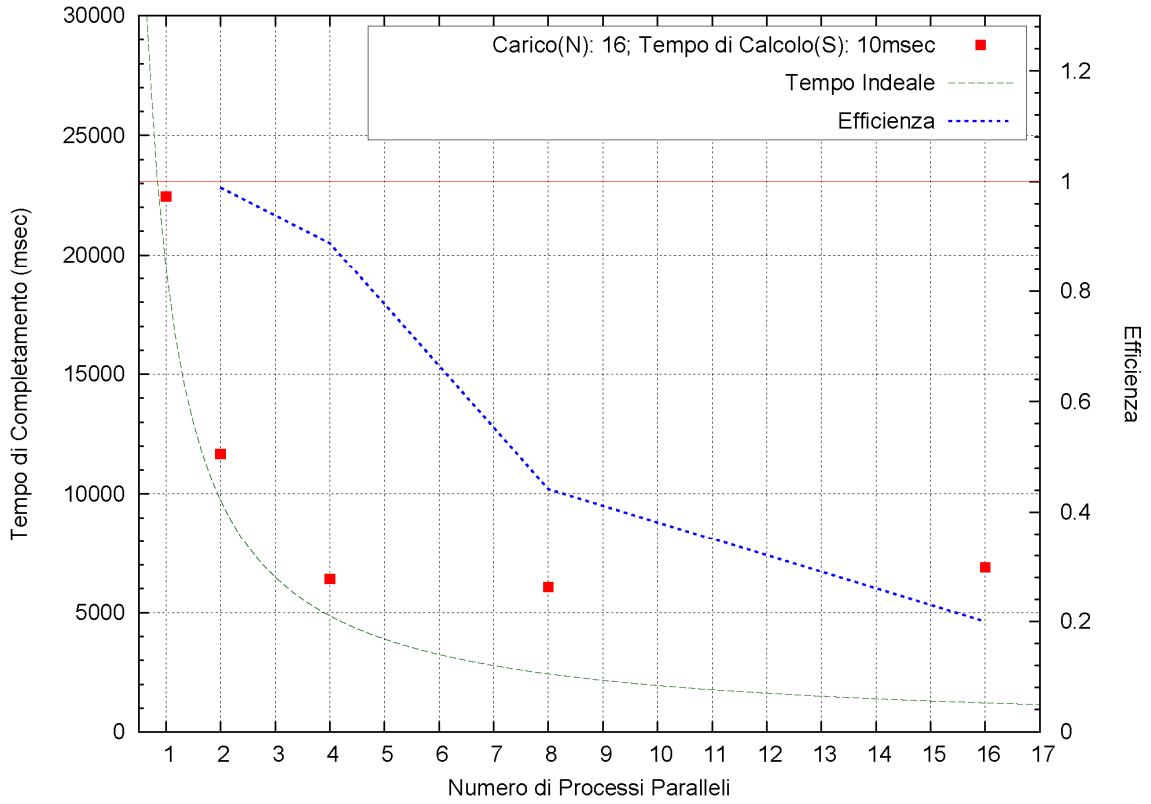

Figura 5.6 - Grana Fine (g= 0,363); N = 16; S = 10msec

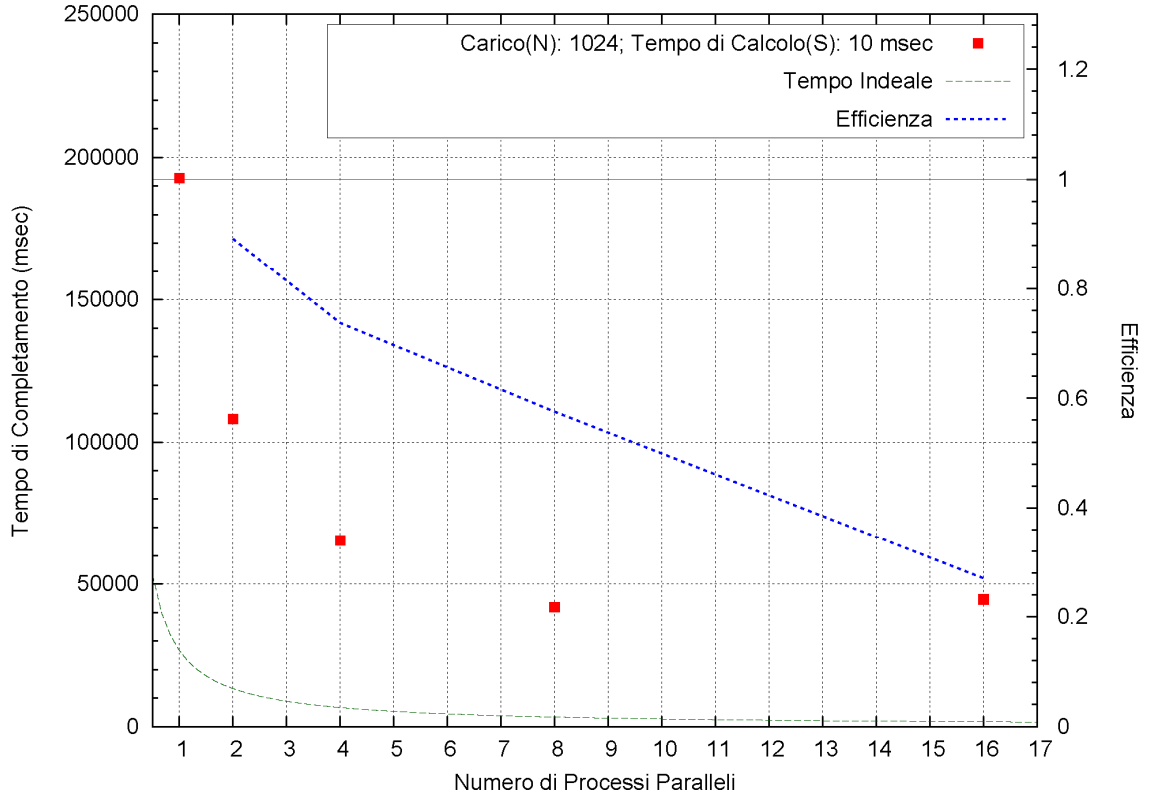

Figura 5.7 - Grana Fine (g= 0,0539); N = 1024; S = 10msec

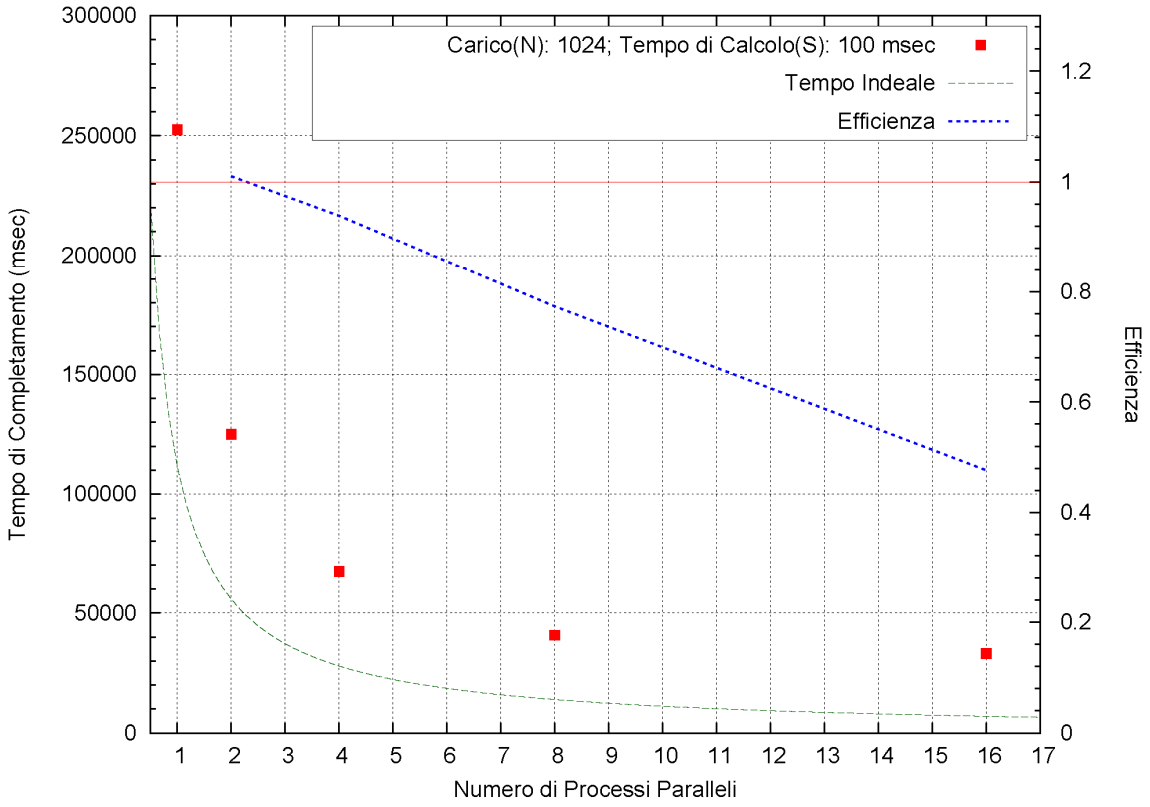

Figura 5.8 - Grana Fine (g= 0,539); N = 1024; S = 100msec

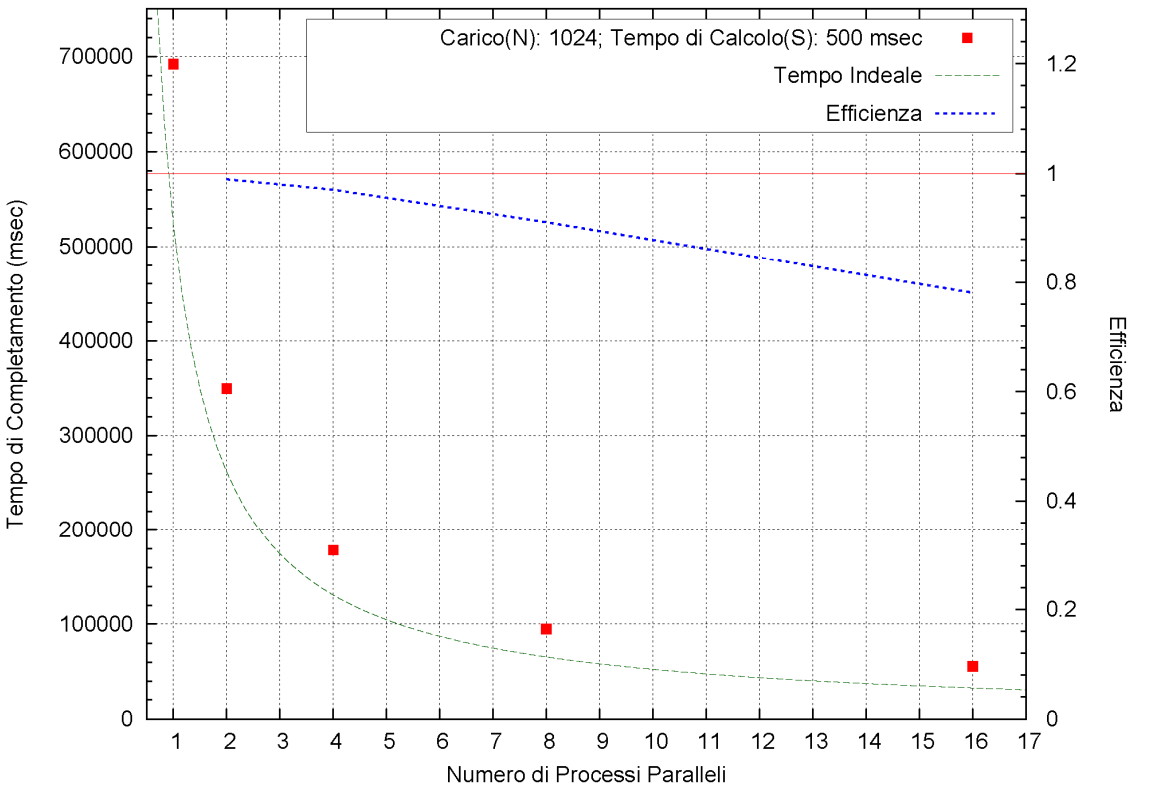

Figura 5.9 - Grana Fine (g= 2,695); N = 1024; S = 500msec

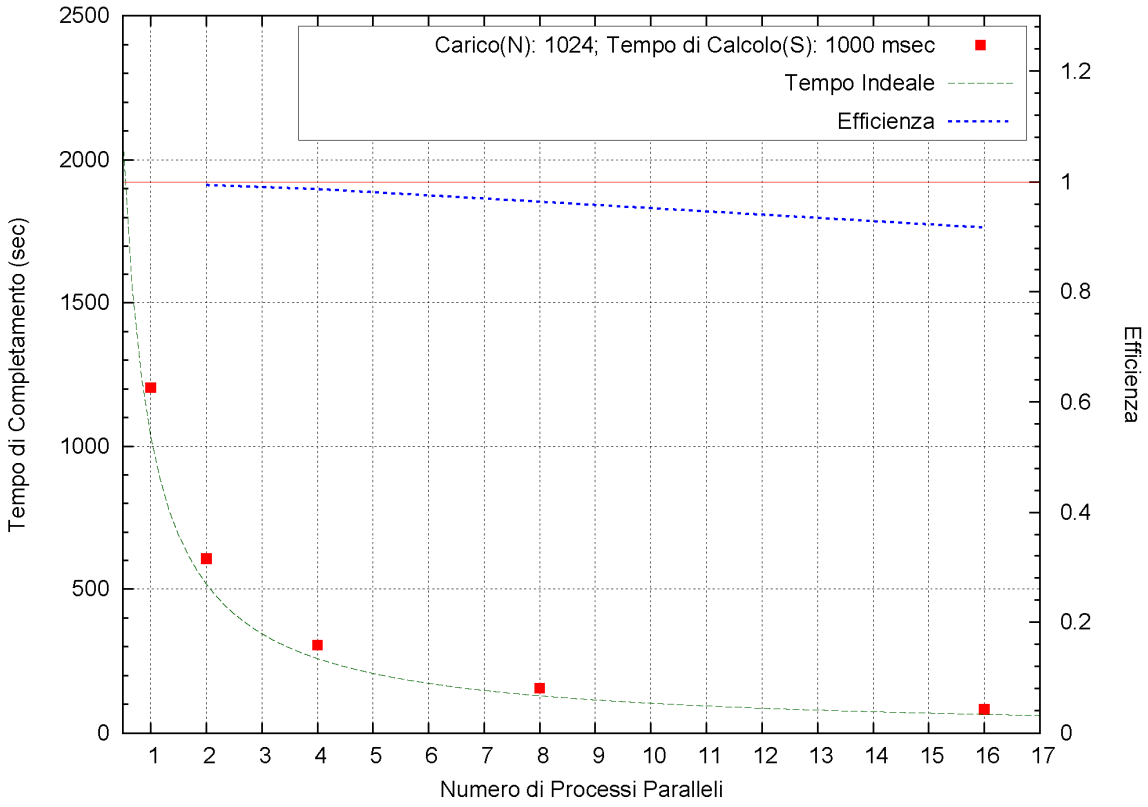

Figura 5.10 - Grana Fine (g= 5,3908); N = 1024; S = 1000msec

I tempi di completamento rilevati risultano notevolmente superiori rispetto ai tempi ideali, per tempi di calcolo estremamente bassi (nell'ordine di 10/100msec); con l'aumentare del parametro S, invece, questi tempi si avvicinano sempre più alla curva ideale.

Questo ci permette di valutare graficamente la scalabilità, in quanto la curva formata dai tempi rilevati dovrebbe avere un andamento parallelo alla curva del tempo ideale, per avere un'ottima scalabilità.

Per controllare la validità del prototipo, non si può dimenticare di valutarne l'efficienza, anch'essa rappresentata negli stessi grafici: all'aumentare del grado di parallelismo l'efficienza cade drasticamente a valori molto bassi per tempi di calcolo nell'ordine di 10/100msec, mentre all'aumentare del tempo di calcolo la curva dell'efficienza tende sempre più ad 1. L'andamento dell'efficienza è una conferma della scalabilità.

Va però detto che si possono considerare risultati soddisfacenti, solamente un'efficienza maggiore di 0,9 al grado di parallelismo 16.

Grana Media

I seguenti scenari per questo tipo di test presentano grana media:

- S = 500ms; N = 8
- S = 500ms; N = 16
- S = 1000ms; N = 8
- S = 1000ms; N = 16
- S = 5000ms; N = 1024

In questo caso, i tempi di comunicazione non influiscono pesantemente sul tempo di completamento in quanto essi sono 10/15 volte più piccoli dei tempi di calcolo.
Le figure 5.11 - 5.15 aiutano nel considerare i risultati ottenuti.

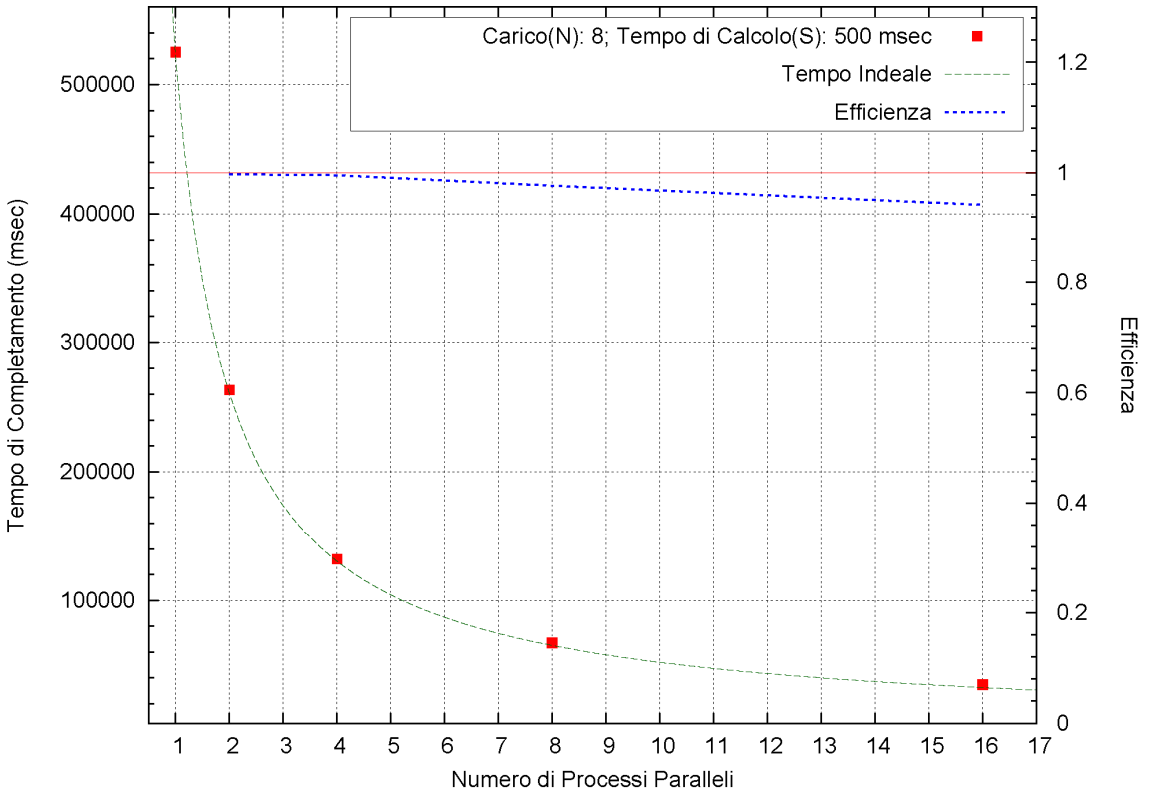

Figura 5.11 - Grana Media (g= 19,607); N = 8; S = 500msec

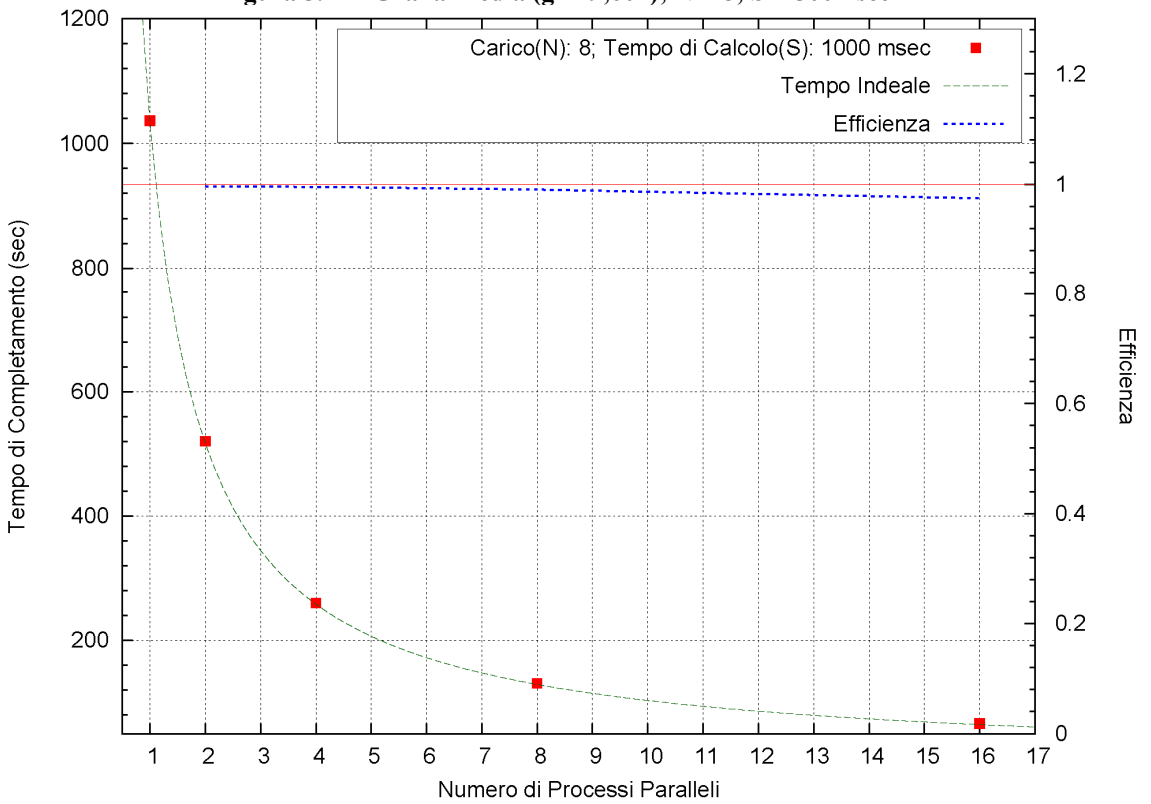

Figura 5.12 - Grana Media (g= 39,215); N = 8; S = 1000msec

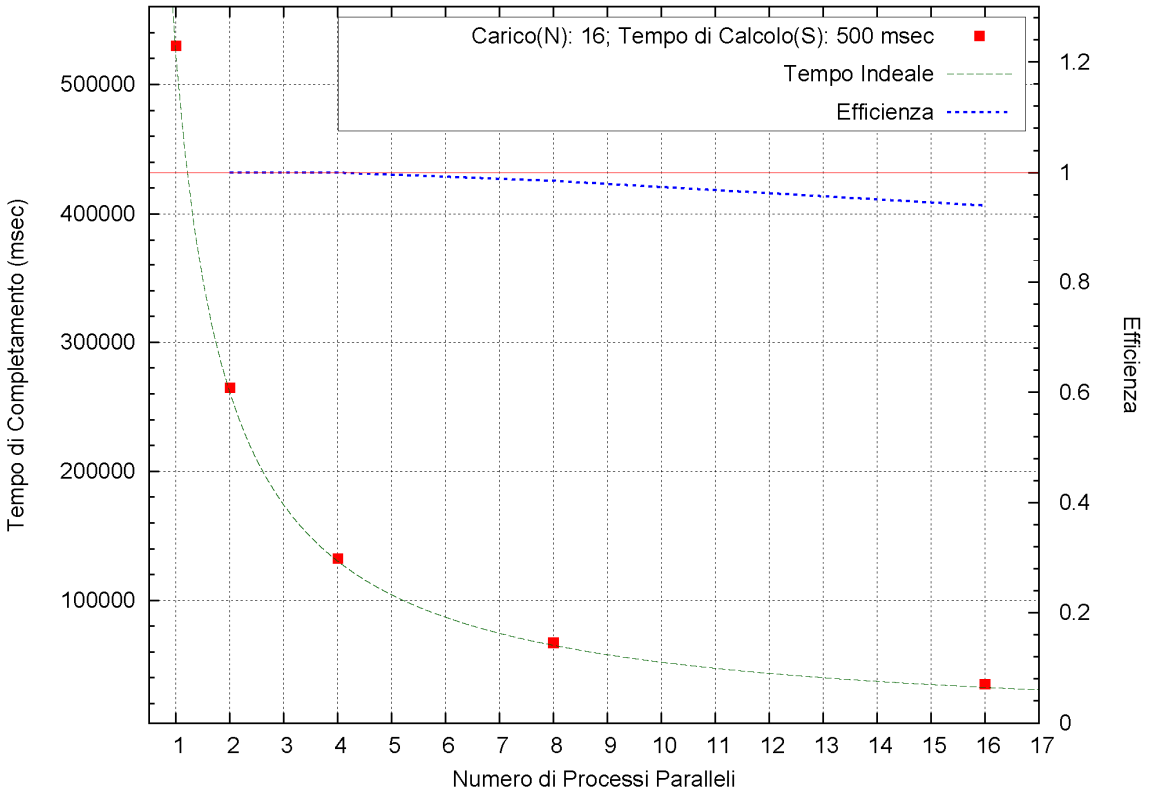

Figura 5.13 - Grana Media (g= 18,181); N = 16; S = 500msec

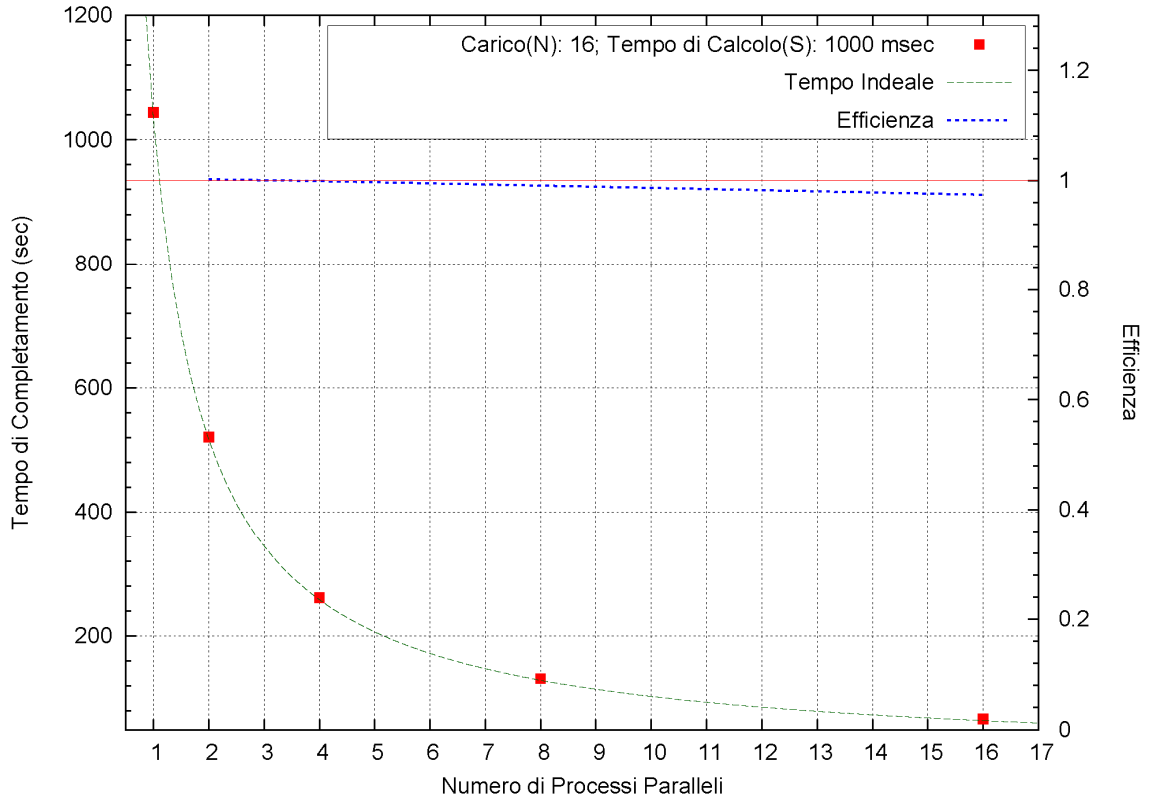

Figura 5.14 - Grana Media (g= 36,363); N = 16; S = 1000msec

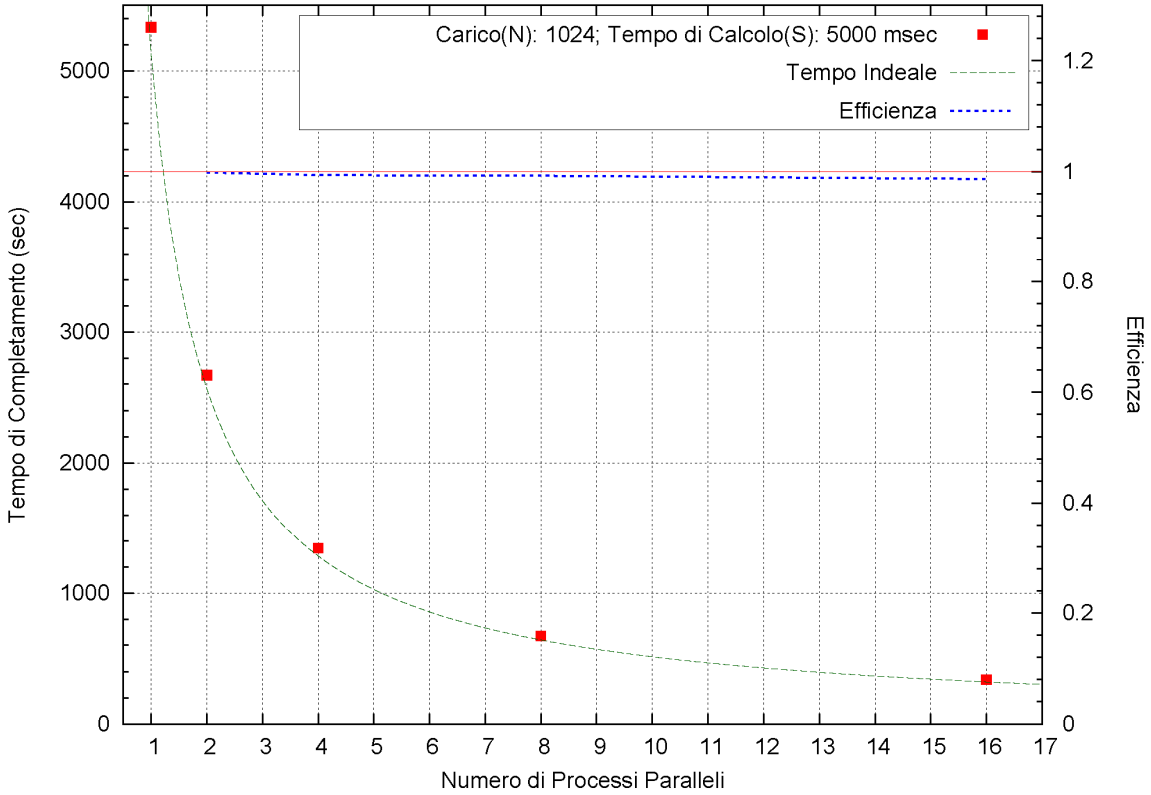

Figura 5.15 - Grana Media (g= 26,954); N = 1024; S = 5000msec

A differenza del calcolo a grana fine, si può notare che in questo caso il tempo di completamento è molto vicino e a volte coincidente con il tempo ideale; questo fatto è anche confermato dall'alta efficienza calcolata, che risulta sempre superiore a 0,92

Grana Grossa

Gli scenari di questo tipo di test presentano grana grossa nei seguenti:

- S = 5000ms; N = 8
- S = 5000ms; N = 16
- S = 10000ms; N = 8
- S = 10000ms; N = 16

In questo caso il tempo di comunicazione è pressoché ininfluente ai fini del calcolo, in quanto è estremamente inferiore al tempo di calcolo.

Le figure 5.16-5.19 aiutano nel considerare i risultati ottenuti.

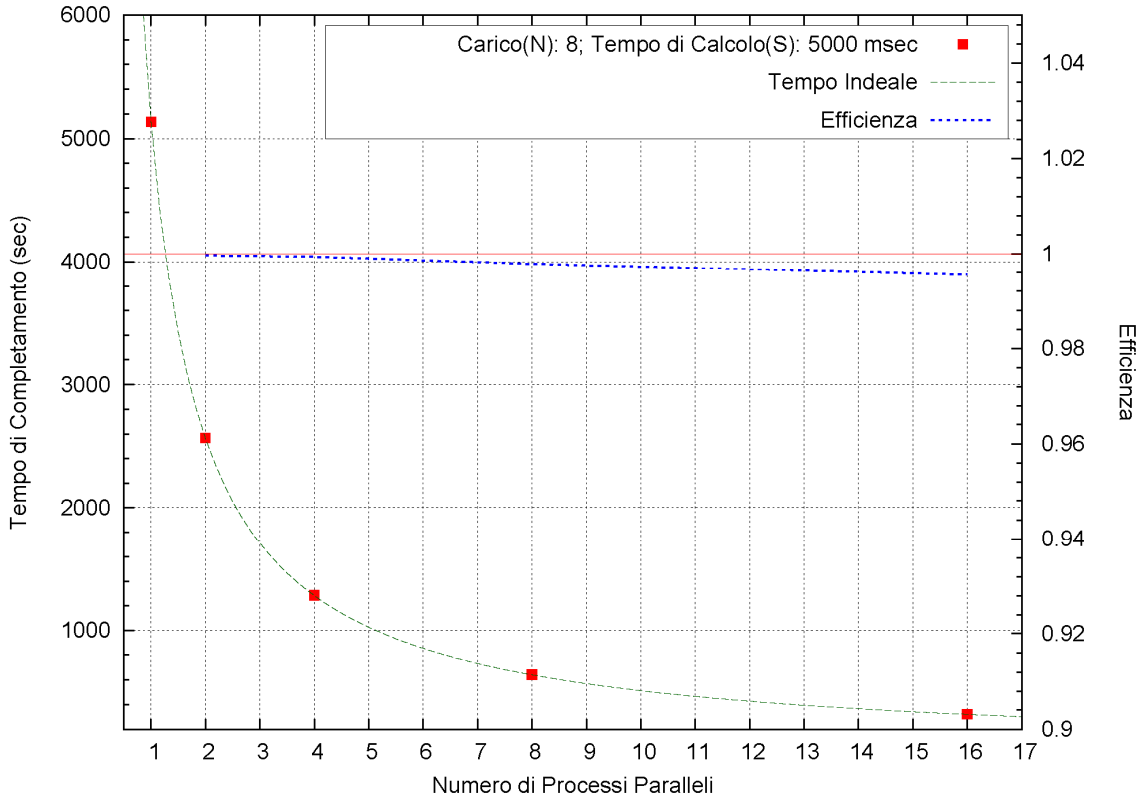

Figura 5.16 - Grana Grossa (g= 196,078); N = 8;S = 5000msec

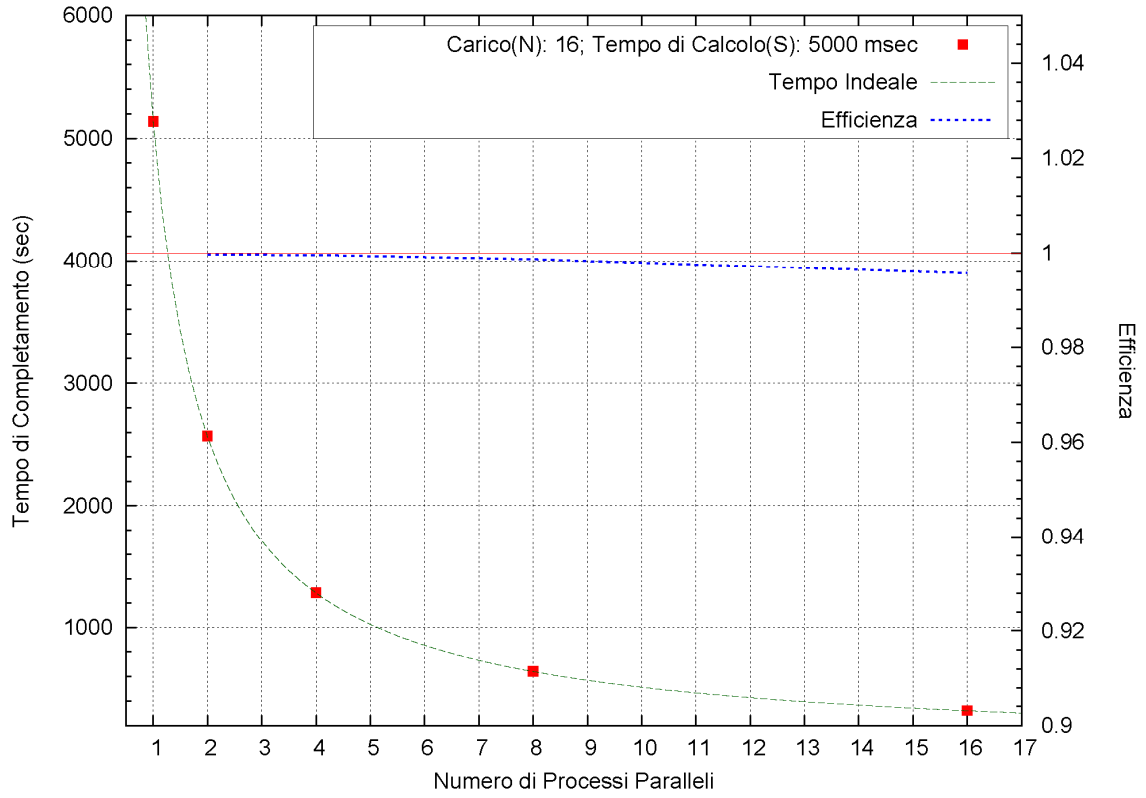

Figura 5.17 - Grana Grossa (g= 181,818); N = 16;S = 5000msec

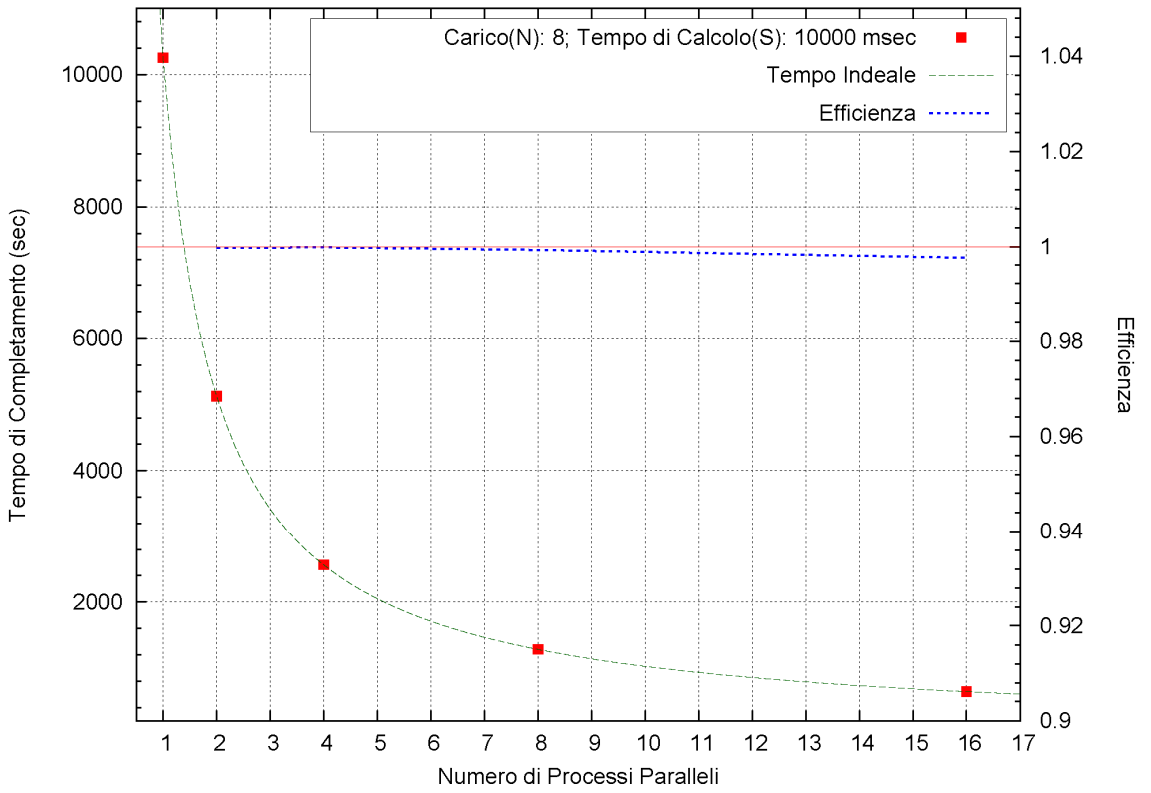

Figura 5.18 - Grana Grossa (g= 392,156); N = 8;S = 10000msec

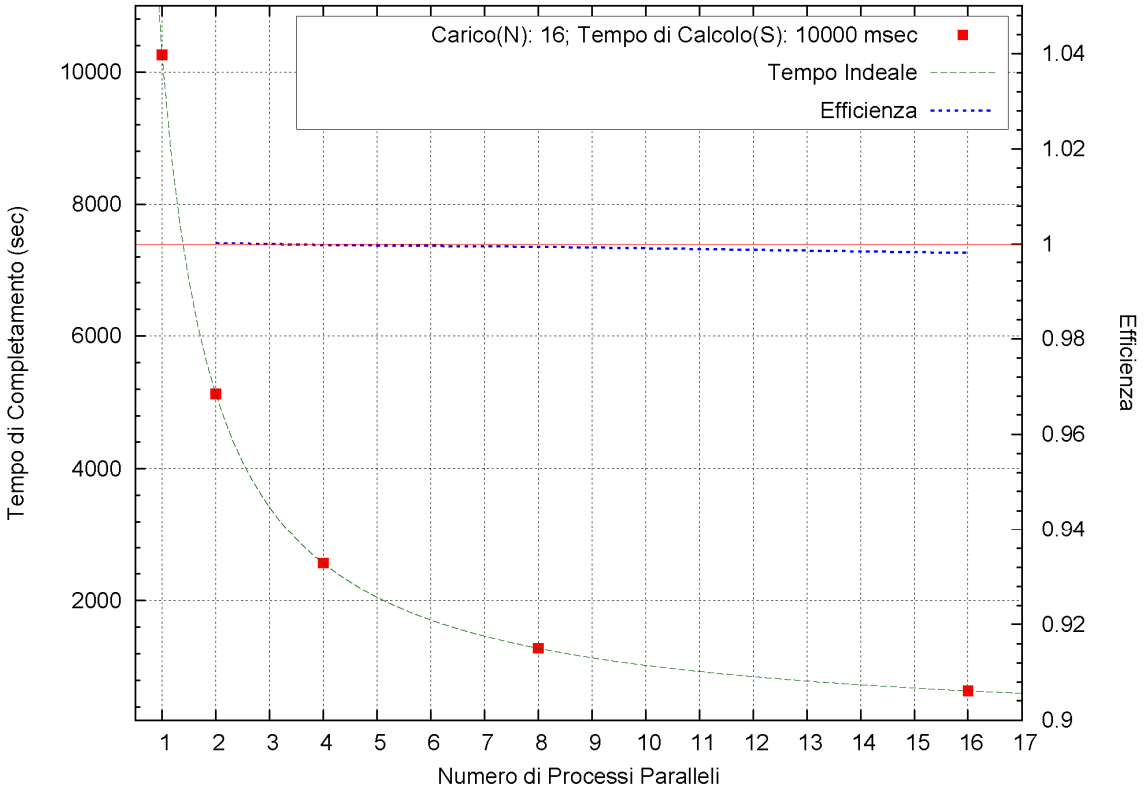

Figura 5.19 - Grana Grossa (g= 363,636); N = 16;S = 10000msec

In questo caso si ha un'efficienza costantemente vicina a 1 (essa varia da 0,99 a 1), mentre la differenza fra il tempo di completamento rilevato e il tempo ideale è sempre assai piccola. È possibile derivare, quindi, una scalabilità ottima in ogni scenario.

5.3.4 Limiti del prototipo

Da un punto di vista globale, l'analisi dei grafici porta a constatare come l'overhead, dovuto all'intero sistema dei Web Services, rappresenti una notevole limitazione alla flessibilità di prestazioni del prototipo.

Infatti il prototipo è estremamente svantaggioso nei calcoli a grana fine. Questo viene anche confermato in letteratura, dove si afferma che i Web Services hanno scarse prestazioni per flussi intensi di dati (Capitolo 2).

Infatti il calcolo a grana fine presenta tempi di completamento assai distanti da quelli ideali, ed inoltre non possiede scalabilità.

Si è constatato che, tenendo costante il tempo di comunicazione, per poter ottenere una performance convincente, è necessario aumentare il tempo di calcolo e di conseguenza la grana. Così è stato possibile mascherare gli effetti dell'overhead.

Infatti si nota che la curva dell'efficienza per ogni dimensione di carico, al variare del parametro S tende sempre più a schiacciarsi verso il valore massimo.

Capitolo 6

Conclusioni

Obiettivo di questa tesi è valutare la possibilità di utilizzare la tecnologia dei Web Services nell'implementazione di programmi per il calcolo parallelo, in remoto, seguendo la stessa metodologia di implementazione utilizzata nella libreria di calcolo parallelo Muskel, basata su skeleton (livello utente) e macro data flow (implementazione). La ragione è che i Web Services offrono una tecnologia duttile e flessibile in quanto hanno come base comune lo standard XML. Questo significa che le funzioni del servizio possono essere implementate in linguaggi diversi di programmazione e che diverse possono essere le piattaforme hardware e software connesse fra di loro. Quindi viene favorito il *loose coupling* e, di conseguenza, l'interoperabilità del *distributed computing* e la scalabilità dei componenti distribuiti.

Si è pensato di elaborare il modello del prototipo e le sue varie parti, articolandole secondo uno schema ergonomico per assolvere il suo compito.

Siamo partiti dalla parte essenziale del Muskel, analizzando le sue funzioni, operando la scelta del paradigma Farm come base del prototipo in quanto presenta i requisiti necessari per la sua realizzazione.

Il modello del prototipo è costituito dal Dispatcher, che ha la funzione del TaskPool di Muskel e dell'Emitter nel Farm, dal Collector, che è stesso del Farm

ed è assimilabile al ResultPool di Muskel e dai Worker, che pubblicano i servizi web.

Nell'implementazione del modello è stato introdotto il Thread, che ha la funzione di inviare la classe di calcolo e i task al proprio servizio, di raccogliere i risultati da esso provenienti e inviarli al Collector per poi riprendere il ciclo fino ad esaurimento dei task.

La distribuzione dei task verso i Thread, che ne fanno richiesta, viene effettuata dal Dispatcher, seguendo la politica FIFO.

Una volta elaborato il prototipo, si è resa necessaria la sua valutazione, che è stata quantizzata mediante dei test e relativi risultati. Quindi, per una valutazione obiettiva dell'efficienza e performance del software nelle sue varie componenti, sono stati scelti dei parametri sulla base dei quali strutturare i test stessi.

Il *test sul tempo di setup* serve a valutare quanto tempo impiega il prototipo a dare inizio al ciclo di distribuzione dei task. Esso varia al variare della dimensione del file della classe di calcolo.

Il *test sul tempo di comunicazione* serve a valutare l'intervallo temporale che intercorre tra l'invio di un array da parte di un thread al servizio e l'inizio della *compute* da parte del servizio stesso. Esso è stato eseguito su tre diverse dimensioni di carico.

Il *test sul tempo di completamento* serve a quantizzare il tempo impiegato dal prototipo per eseguire tutti i task designati con tre valori del parametro `N`, che rappresenta la dimensione del carico, sei valori del parametro `S`, che indica il tempo di calcolo e con diversi gradi di parallelismo. Il risultato è considerato accettabile se l'efficienza è superiore al 90% ad un grado di parallelismo pari a 16.

Facendo il bilancio dei risultati ottenuti è emerso che i risultati per grane medio/alte sono stati ottimi; di contro, per grane fini i risultati si sono rivelati scarsi, ma in considerazione di N alto ed S non basso, potevano essere accettabili.

Infatti per quanto riguarda le grane fini, analizzando i grafici si nota che la curva dei tempi di completamento non segue l'andamento della curva del tempo ideale, distaccandosi notevolmente da questa, ma addirittura in alcuni scenari tende a risalire leggermente con un grado di parallelismo superiore ad 8. Quanto detto, però, non vale per lo scenario [N = 1024, S = 1000msec], in cui la curva dei tempi di completamento risulta essere abbastanza fedele all'andamento di quella ideale, garantendo così una scalabilità molto buona e un'efficienza del 94% ad un grado di parallelismo pari a 16.

Dall'analisi di tutti i risultati ottenuti, è emerso che quindi è necessario aumentare il tempo di calcolo, per poter ottenere dei risultati ottimi in termini di scalabilità ed efficienza, tenendo fisso il tempo di comunicazione; questo vuol dire, in definitiva, che è necessario aumentare la grana del calcolo.

La spiegazione di questa discrepanza è da ricercare nella ingombrante presenza dell'overhead, legato in gran parte alle attività di serializzazzione/deserializzazione dei parametri trasferiti alle/dalle macchine remote per il calcolo delle macro istruzioni data flow, che per grane medio/alte viene mascherata.

Considerando, poi, il test sul tempo di setup, la curva dei risultati si avvicina molto ad un andamento lineare nell'intervallo compreso tra i 100KB e gli 800KB, mentre agli estremi è mantenuta la linearità, ma con un coefficiente angolare minore; per cui nei punti in cui l'ascissa assume i valori 100KB e 800KB la curva presenta un cambio di direzione sensibile. Se ne deduce, quindi, che il tempo di setup risulta variabile per file di piccola e grande dimensione.

In sintesi si può dire che questo prototipo scala per flussi non intensi di dati, come è confermato in letteratura.

Le limitazioni che presenta il prototipo sono date dal fatto che esso può fare solo calcoli con tipi di dati specifici e non generici, come si era deciso all'inizio del lavoro. Si sono, infatti, incontrate difficoltà legate agli strumenti usati e in particolar modo al *databinding*, in quanto i Web Services sono stati pensati per i programmi, e non per i programmatori.

Si sarebbe dovuto usare a mano i meccanismi e i protocolli per l'implementazione dei servizi e dei client. Questo, però, era lungo e complicato perché le librerie dei framework sono sostanzialmente automatiche o, al contrario, prolisse e non intuitive, talché era facile incorrere nell'errore. Queste limitazioni sono state sostanziate dalla scarsità di tempo a disposizione.

Da questa tesi emergono alcuni sviluppi possibili:

- l'estensione del prototipo per trattare tipi di dati generici
- l'introduzione di caratteristiche di sicurezza nell'esecuzione dell'operazioni.
- completa integrazione nel prototipo attuale di Muskel.

L'estensione del prototipo è una diretta conseguenza della limitazione sopra descritta.

Un aspetto che non è stato preso in considerazione in questa tesi è la sicurezza; infatti non sono stati previsti controlli durante l'accesso al filesystem sia durante la fase di *load* sia durante quella di *compute*. Il pericolo risiede nel fatto che senza delle restrizioni adeguate, i dati sono facilmente accedibili . Un'ulteriore mancanza di sicurezza è legata alla natura del calcolo stesso, in quanto la *compute* potrebbe eseguire un'eventuale codice dannoso.

Uno sviluppo successivo potrebbe essere quello di integrare completamento il prototipo nel Muskel, realizzandone una versione che utilizzi i Web Services al posto di RMI.

Con un maggior tempo a disposizione è possibile fare una ricerca capillare di strumenti che risultino più efficienti nel loro utilizzo e che, quindi, permettano di assolvere in maniera completa all'obiettivo.

APPENDICE A

Codice Sorgente

Di seguito viene riportata la parte principale del codice sorgente che realizza l'implementazione del prototipo. L'elaborazione di questo codice, scritto nel linguaggio di programmazione Java, ha richiesto la configurazione degli strumenti utilizzati per la realizzazione di questa tesi. Infatti, sviluppato il codice, per il suo utilizzo come Web Service è stato seguito un iter che ha portato il codice alla pubblicazione del servizio.

Vengono esaminati i singoli file, contenenti ciascuno una sola classe e , per questione di chiarezza, questi saranno mostrati in un ordine che si riferisce alla figura 4.1, guardata da destra verso sinistra :

1. Il web service
    - **1.1.** (LoadAndCompute.java)
    - **1.2.** (services.xml)
2. Il thread (WSThread.java)
3. Il Dispatcher (Dispatcher.java)
4. Il Collector (Collector.java)
5. La classe di Calcolo (IncrArray.java)
6. Il main (Client.java)

A.1 Il web service

Qui di seguito viene riportato il codice sorgente che realizza il servizio web del prototipo.

La pubblicazione di un web service in Axis2 necessita di un archivio ".aar"contenente il bytecode delle classi che implementano il servizio e del relativo file "services.xml".

A.1.1 LoadAndCompute.java

La classe contenuta in questo file attua il servizio del prototipo, mette a disposizione le due operazioni *load* e *compute*.

```
import java.net.*;
import java.io.*;
import java.lang.ClassLoader;
import java.lang.reflect.*;
import java.util.*;

public class LoadAndCompute{
	private static String path = "/ws/";
	private static File file = new File(path);

	/**
	* WebService load
	* @param : data = array di byte contenente il file da salvare
	* @param : name = nome del file da salvare
	* @return : true se tutto e' andato a buon fine
	**/
	public boolean load(byte[] data,String name)throws IOException{
		OutputStream os = new FileOutputStream(path+name);
		int off=0;
		os.write(data,off,data.length-off);
		os.close();
```

```
		File prova = new File (path+name);
		if(prova.exists() && prova.isFile() )
			return true;
		else return false;
	}

	/**
	* WebService compute
	* @param : classe = nome della Classe del Calcolo
	* @param : inputs = array di input alla classe del calcolo
	* @param : inJar = nome del file jar che contiene
	* il file class della Classe del Calcolo,
	* oppure la parola-chiave "no",
	* se la classe non e' in un jar
	* @return : un array con i risultati del Calcolo
	**/
	public Long[] compute(String classe,Long [] inputs,String inJar){
		//classe e' senza il .class
		Object ris=null;URL [] urls;ClassLoader cl;
		if(classe.endsWith(".class"))
			classe = classe.substring(0,classe.indexOf(".class"));

		if(inJar.equals("no")){
			try{
				urls = new URL[]{file.toURL()};
				cl = new URLClassLoader(urls);
			}catch(MalformedURLException e){
				System.out.println("Errore MalformedURL");
				return (new Long[]{new Long(10000)});
			}catch(Exception e){
				System.out.println("Errore generico.\n"+e);
				return (new Long[]{new Long(20000)});}
		}else{
			try{
				String jjj=path+inJar;
				File fJar=new File(jjj);
				urls = new URL[]{fJar.toURL()};
				cl = new URLClassLoader(urls);
			}catch(MalformedURLException e){
				System.out.println("Errore MalformedURL");
				return (new Long[]{new Long(10000)});
			}catch(Exception e){
```

```
                        System.out.println("Errore generico.\n"+e);
                        return (new Long[]{new Long(20000)});}
                }
                try{
                        Class cls = cl.loadClass(classe);
                        Object a = cls.newInstance();
                        Vector input=new Vector(inputs.length);
                        for (int i =0; i<inputs.length;i++){
                                input.add(inputs[i]);
                        }
                        //Metodo void setParam(Vector<Long> a)
                        Class[] par = new Class[]{ Vector.class };
                        Method setParam = cls.getMethod("setParam",par);
                        Object[] parametro=new Object[]{input};
                        setParam.invoke(a,parametro);

                        //Metodo Vector<Long> compute()
                        Class [] parC = new Class[]{};
                        Method compute = cls.getMethod("compute",parC);
                        ris = compute.invoke(a,new Object[]{});

                }catch (ClassNotFoundException e) {
                        System.out.println("Errore ClassNotFound.");
                        return (new Long[]{new Long(30000)});
                }catch(Exception e){
                        System.out.println("Errore generico.\n"+e);
                        return (new Long[]{new Long(40000)});}

                Vector v = (Vector)ris;
                Long [] ret = new Long[v.size()];
                v.copyInto(ret);
                return ret;
        }
}
```

A.1.2 Services.xml

```
<service name="LoadAndCompute">
   <parameter name="ServiceClass" locked="false" > LoadAndCompute
   </parameter>
      <parameter name="enableMTOM" locked="false">true</parameter>
      <messageReceivers>
            <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-
out" class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
      </messageReceivers>
</service>
```

A.2 Il thread (WSThread.java)

Questa classe crea il thread preposto alla gestione dell'elaborazione dei task del Dispatcher. Per questione di leggibilità, non viene riportato il codice relativo allo stub di supporto al thread, in quanto esso è stato generato automaticamente.

```
package org.apache.ws.axis2;

import java.io.*;
import org.apache.ws.axis2.LoadAndComputeStub.*;
import javax.activation.*;
import org.apache.axis2.Constants;

public class WSThread extends Thread{

      String endpoint=null, classe=null, jarr=null;
      Dispatcher disp;
      Collector coll;
      long[] input;
      int N=0, taskCorrente;
      long S=0;
```

```
	LoadAndComputeStub stub;

	public WSThread(String targetEndpoint, String fjar,
		String cl,Dispatcher d,Collector c, int dimArray,
		long attesa){
		endpoint = targetEndpoint;
		disp = d;
		coll=c;
		N=dimArray;
		S=attesa;
		jarr = fjar;
		classe=cl;
		taskCorrente=0;
		try{
			stub = new LoadAndComputeStub(endpoint);
// Abilito MTOM
stub._getServiceClient().getOptions().setProperty(Constants.Configuration
.ENABLE_MTOM, Constants.VALUE_TRUE);
//Aumento il tempo di TimeOut
stub._getServiceClient().getOptions().setTimeOutInMilliSeconds(10000);
		}catch(Exception e){System.out.println(e);}

	}//cstor
	public void run(){
		//carico la classe
		boolean ok=true;long start=0,stop=0;float dim=0,fd=0;File f;
		try{
		if(jarr.equals("no")){
			ok = loadClass(classe);
			f=new File(classe);
		}else{
			ok = loadClass(jarr);
			f=new File(jarr);
		}
		if(!ok){
			System.out.println("Thread "+this.getName()+": errore
nel caricamento della classe!");
			return;
		}

		while (! disp.finito()){
			//prendo il task e lo metto in un long[]
			long[] input = disp.getNext();
```

```
                    //gli elementi da 0-(N-1)  sono i dati di input
                    //l'elemento N e' il numero del task;
                    //salvo in taskCorrente l'ultimo elemento dell'array
                    taskCorrente = (int)input[N];
                    //metto nell'ultimo elemento dell'array S
                    input[N]=S;
                    //calcolo con computes(...)
                    //prendo il risultato e lo metto in un altro long[]
                    long[]res = computes(classe,input, jarr);
                    //richiamo coll.setReturn(array, taskCorrente)
                    ok = coll.setReturn(res,taskCorrente);
                    if(!ok){
                    System.out.println("Thread "+this.getName()+": errore
nel depositare il risultato del task #"+taskCorrente+"!");
                    }
             }//while
             }catch(Exception e){System.out.println(e);}
      }//run

   boolean loadClass(String classe)throws Exception{
      LoadAndComputeStub.Load request = new LoadAndComputeStub.Load();
             FileDataSource fds=new FileDataSource(classe);
             DataHandler dh=new DataHandler(fds);
             request.setParam0(dh);
             classe = getClassFile(classe);
             request.setParam1(classe);
            //Invoke the service
             LoadResponse response = stub.load(request);
             return response.get_return();
      }

 long[] computes(String classe,long[] ve,String jarr)throws Exception{
      LoadAndComputeStub.Compute req2 = new LoadAndComputeStub.Compute();
             classe = getClassName(classe);
             req2.setParam0(classe);
             req2.setParam1(ve);
             jarr = getClassFile(jarr);
             req2.setParam2(jarr);
             ComputeResponse risp = stub.compute(req2);
             return risp.get_return();
      }
```

```
String getClassName(String c){
	String nome = getClassFile(c);
	if(nome.endsWith(".class"))
		nome = nome.substring(0,nome.indexOf(".class"));
	return nome;
}
String getClassFile(String c){
	String nome=null;
	if(c.contains("/")){
		nome = c.substring(c.lastIndexOf("/")+1);
		return nome;
	}else{ return c;}
}
```

A.3 Il Dispatcher (Dispatcher.java)

Il Dispatcher distribuisce i task da eseguire ai thread che ne fanno richiesta.

```
package org.apache.ws.axis2;

import org.apache.ws.axis2.*;
import java.util.*;

public class Dispatcher {

	private int m; //Numero di array, o di task
	private int count; //Numero di array "prelevati"
	private int N; //Dimensione del "carico"

	public Dispatcher(int numTask, int dimArray){
		m=numTask;
		count=0;
		N=dimArray;
	}

	//Testa se sono terminati i task
	public synchronized boolean finito(){
		if(count == m)
			return true;
		else
			return false;
	}//finito
```

```
	/**
	* Method getNext
	* @return : un array di N+1 elementi dove:
	* gli elementi da 0-(N-1)  sono i dati di input
	* l'elemento N e' il numero del task
	**/
	public synchronized long[] getNext(){
		count++; //il primo task e' 1.
		long [] a = new long[N+1];
		for(int i=0; i<N;i++){
			//creo l'array da count*10 fino a count*10+(N-1)
			a[i]=(count*10)+i;
		}
		a[N]=(long)count;
		return a;
	}
}
```

A.4 Il Collector (Collector.java)

Il Collector raccoglie i risultati dei task eseguiti dai thread, ordinandoli.

```
package org.apache.ws.axis2;

import java.util.*;

public class Collector {

	int m;  //Numero di array, o di task
	private int count; //Numero di array "raccolti"
	private int N; //Dimensione del "carico"
	//Struttura per contenere la risoluzione della computazione
	private long [][] risolti;

	public Collector(int numTask, int dimArray){
		m=numTask;
		N=dimArray;
		count=0;
		risolti = new long [m][N];

	}
	//Testa se ha raccolto tutti i dati
	public synchronized boolean finito(){
		if(count == m){
			return true;
		}else{
			return false;
		}
	}
```

```
	/**
	* Method setReturn
	* @param : r e' formato da N elementi dove:
	* gli elementi da 0-(N-1) sono i dati calcolati
	* @return : true se il salvataggio è andato a buon fine
	**/
	public synchronized boolean setReturn(long [] r, int task){
		//posizione del task all'interno della matrice
		int arrTask = task -1;
		for (int i=0; i<N;i++){
			risolti[arrTask][i]=r[i];
		}
		count++;
		return ok;
	}

	//richiamato solo x controllare l'esattezza del calcolo
	public String printSolution(){
		String ritorno="";
		for(int i=0;i<m;i++){
			ritorno += (i+") ");
			ritorno+= java.util.Arrays.toString(risolti[i]);
			ritorno+="\n";
		}
		return ritorno;
	}
}
```

A.5 La classe di Calcolo (IncrArray.java)

IncrArray è la classe del Calcolo usata nei test valutativi.

```
import java.util.Vector;
import java.lang.*;

public class IncrArray{

	long [] parametro;
	long attesa=0; //attesa è il parametro S

	/**
	* Method setParam
	* @param : Vector a contiene N+1 elementi dove:
	* g li elementi da 0 a N-1 sono i dati di input
	* l'ultimo elemento è il tempo di attesa S
	**/
	public void setParam(Vector a){
		Object[]b=a.toArray();
		parametro = new long[b.length-1];
```

```
            for(int i =0;i<b.length-1;i++){
                  parametro[i]=((Long)(b[i])).longValue() ;
            }

            attesa = ((Long)(b[b.length-1])).longValue();
      }
      /**
      * Method compute
      * @return : un Vector a contiene N+1 elementi dove:
      * g li elementi da 0 a N-1 sono i dati di input
      * l'ultimo elemento è il tempo di attesa S
      **/

      public Vector compute(){
            for(int i=0;i<parametro.length;i++){
                  parametro[i]+=1;
            }
            Vector v = new Vector(parametro.length);
            for(int i=0;i<parametro.length;i++){
                  v.add(parametro[i]);
            }
```

A.6 Il main (Client.java)

Questa classe contiene il metodo main, che crea i thread, il dispatche e il collector e da' inizio all'elaborazione del calcolo.

```
package org.apache.ws.axis2;

import org.apache.ws.axis2.Collector;
import org.apache.ws.axis2.Dispatcher;
import org.apache.ws.axis2.WSThread;

public class Client {

    public static void main(String[] args) throws Exception {

          long S; int N;
          int m=0,n=0;
          String classe, fileJar;
          Thread []threads;

          if(args.length <6){
      System.out.println("Usage: Client {fileJar|no} Classe
numeroDiArray(m) numeroDithread(n) dimArray(N) tempoDiAttesa(S)");
                return;
          }else{
                fileJar = args[0];
                classe = args[1];
```

```
                m=Integer.parseInt(args[2]);
                int nn= Integer.parseInt(args[3]);
                threads = new Thread[nn];
                n=nn;
                N=Integer.parseInt(args[4]);
                S=Long.parseLong(args[5]);
          }

          Dispatcher disp = new Dispatcher(m,N);
          Collector coll = new Collector(m,N);

          for(int i=0;i<n;i++){
                String epr ;
                if( (i+4)==15 ){
             epr = "http://u20:8080/axis2/services/LoadAndCompute";
                   }else{
      epr = "http://u"+(i+4)+":8080/axis2/services/LoadAndCompute";}
      threads[i]=new WSThread(epr,classe,disp, coll, N,S);
          }

      System.out.println("Calcolare "+m+" array di "+N+" elementi con
"+n+" thread paralleli e un tempo di attesa per calcolo di "+S+" ms con
la classe "+getClassName(classe)+".");

      for(int i=0;i<n;i++){
            threads[i].start();
      }
      for(int i=0;i<n;i++){
            threads[i].join();
      }

      System.out.println("\nCalcolo effettuato.\n");
      System.out.println(coll.printSolution());
   }


   static String getClassName(String c){
         String nome = getClassFile(c);
         if(nome.endsWith(".class"))
               nome = nome.substring(0,nome.indexOf(".class"));
         return nome;
   }
   static String getClassFile(String c){
         String nome=null;
         if(c.contains("/")){
               nome = c.substring(c.lastIndexOf("/")+1);
               return nome;
         }else{ return c;}

   }
}
```

Bibliografia

[01] W3C Web Services Activity home page , 23 Maggio 2007
http://www.w3.org/2002/ws/

[02] W3C Recommendation - *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, 27 Aprile 2007
http://www.w3.org/TR/soap12-part1/

[03] Muskel (sito web), 28 Maggio 2006
http://www.di.unipi.it/~marcod/Muskel/index.html

[04] Heather Kreger – "*Web Services Conceptual Architecture" (WSCA 1.0)* - IBM Software Group, Maggio 2001

[05] W3C Working Group – "*Web Services Architecture" (WSA 04)* - Note 11 Febbraio 2004

[06] W3C Note - *Web Services Description Language (WSDL) 1.1* , 15 Marzo 2001
http://www.w3.org/TR/wsdl

[07] IBM: developerWorks - *SOA & Web services zon*e, 25 May 2007
http://www-128.ibm.com/developerworks/soa

[08] Hao He – "*What is Service-Oriented Architecture?"* - O' Reilly Press, 30 Settembre 2003

[09] W3C XML homepage, 08 Maggio 2007
http://www.w3.org/XML/

[10] W3C Working Draft - *Web Services Glossary*, 14 Novembre 2002
http://www.w3.org/TR/2002/WD-ws-gloss-20021114/

[11] Joseph Williams –" *The Web services debate: J2EE vs. .NET*" - Communications of the ACM, Volume 46, Issue 6, 2003

[12] W3C XML Schema home page, 23 Gennaio 2007
http://www.w3.org/XML/Schema

[13] W3Schools - *SOAP Tutorial*, dopo 08 Maggio 2000
http://www.w3schools.com/soap/default.asp

[14] OASIS - *UDDI specifications*, 2 Novembre 2006
http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[15] BEA - *WS-Transaction specification*, 30 Gennaio 2004
http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html

[16] IBM - *WS-Security Specification*, 01 Marzo 2004
http://www-128.ibm.com/developerworks/library/specification/ws-secure/

[17] WS-I *Basic Security Profile*, 30 Marzo 2007
http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

[18] WS-I *Basic Profile 1.0 Specification*, 16 Aprile 2004
http://www.ws-i.org/Profiles/BasicProfile-1.0.html

[19] OASIS - *WS-Reliability 1.1 specification*, 15 Novembre 2004
http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

[20] OASIS - *WS-ReliableMessaging 1.1 Committee Specification*, 11 Aprile 2007
http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-cs-01.pdf

[21] IBM - *WS-Addressing specification*, 10 Agosto 2004
http://www-128.ibm.com/developerworks/library/specification/ws-add/

[22] OASIS - *WS-Coordination specification*, Aprile 2007
http://docs.oasis-open.org/ws-tx/wscoor/2006/06/

[23] W3C Recommendation - *SOAP Message Transmission Optimization Mechanism*, 25 January 2005
http://www.w3.org/TR/soap12-mtom/

[24] W3C Recommendation - *XML-binary Optimized Packaging, W3C Recommendation*, 25 January 2005
http://www.w3.org/TR/xop10/

[25] Donald Ferguson, Tony Storey, Brad Lovering, John Shewchuk - *"Secure, Reliable, Transacted Web Services Architecture and Composition"*, IBM developerWorks 28 Oct 2003
http://www.ibm.com/developerworks/webservices/library/ws-securtrans/

[26] Windows Communication Foundation, 19 Aprile 2007
http://wcf.netfx3.com/

[27] InnoQ - *Web Services Standards - An overview of the Web services standards landscape*, 23 febbraio 2007
http://www.innoq.com/soa/ws-standards/

[28] Sun's Java EE page, 25 maggio 2007
http://java.sun.com/j2ee/

[29] Jit Ghosh - *".NET vs. J2EE - Compare & Contrast "* - Microsoft Corporation, 5 Marzo 2002

[30] Gunjan Samtani and Dimple Sadhwani – "*Web Services and Application Frameworks*", 8 Aprile 2002
http://www.webservicesarchitect.com/content/articles/samtani04.asp

[31] IBM WebSphere Application Server, 22 Maggio 2007
http://www-306.ibm.com/software/webservers/appserv/was/

[32] The JWSDP Developer Community, 7 Aprile 2006
https://jwsdp.dev.java.net/

[33] Massimiliano Bigatti – *"Web Services più sicuri e veloci"* - MokaByte 91- Dicembre 2004
http://www.mokabyte.it/2004/12/jwsdp_1-5.htm

[34] Sun J2SE, Maggio 2007
http://java.sun.com/javase/

[35] The Apache Software Foundation - *Apache Tomcat* , 14 Maggio 2007
http://tomcat.apache.org/

[36] The Apache Software Foundation – *Apache Axis2/Java*, 27 Aprile 2007
http://ws.apache.org/axis2/

[37] The Apache Software Foundation – *Web Service Axis*, 6 maggio 2006
http://ws.apache.org/axis/

[38] The Apache Software Foundation – *Apache Axiom*, 21 aprile 2007
http://ws.apache.org/commons/axiom/index.html

[39] JaxMe Open Source Project, 31 agosto 2005
http://ws.apache.org/jaxme/

[40] JiBX: Binding XML to Java Code, 23 maggio 2007
http://jibx.sourceforge.net/

[41] The Apache Software Foundation – *XMLBeans*, 25 Luglio 2007
http://xmlbeans.apache.org/

[42] Java Architecture for XML Binding (JAXB)
http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[43] The Apache Software Foundation – *Apache Sandesha2*, 8 maggio 2006
http://ws.apache.org/sandesha/sandesha2/index.html

[44] The Apache Software Foundation – *Securing SOAP Messages with Rampart*, 04 May 2007
http://ws.apache.org/axis2/modules/rampart/1_2/security-module.html

[45] Pianosa presso il Dipartimento di Informatica dell'Università degli Studi di Pisa, 4 maggio 2004
http://pianosa.di.unipi.it/

[46] Marco Zanneschi – "*Appunti di Architetture Parallele e Distribuite*" – Dipartimento d'Informatica, SEU Pisa, 2001-2002

[47] FOLDOC – Free On Line Dictionary Of Computing, 08 Maggio 1997
http://foldoc.org/foldoc.cgi?granularity